




G R A P H I C S FOR THE MACINTOSH™



AN IDEA BOOK

JOHN P. GRILLO & J. DOUGLAS ROBERTSON

CBS Computer Books



GRAPHICS FOR THE MACINTOSH

An Idea Book

Mexico City Rio de Janeiro Madrid

Apple®, Macintosh™, MacPaint™ and MacWrite™ are registered trademarks of the Apple Computer Corp.

TRS-80® is a registered trademark of the Tandy Corp.

MacWorld™ is a registered trademark of PC World Communications.

Microsoft® is a registered trademark of the Microsoft Corp.

IBM® is a registered trademark of International Business Machines Corporation.

Generic Computing™ is a registered trademark of the Generic Computing Co., Inc.

HP-150 is a registered trademark of Hewlett-Packard Corp.

Star Wars is a registered trademark of Lucas Films

Commodore 64™ is a registered trademark of Commodore Business Machines.

Acquisitions Editor: Deborah L. Moore

Production Manager: Paul Nardi

Composition: The Publisher's Network

Cover Design: Anthony Frizano

Illustrations: Grillo and Robertson

First distributed to the trade in 1985 by Holt, Rinehart and Winston
General Book Division

Copyright© 1985 CBS College Publishing

All rights reserved.

Address correspondence to:

383 Madison Avenue, New York, NY 10017

Library of Congress Cataloging in Publication Data

Grillo, John P.

Graphics for the Macintosh.

(CBS Computer Books)

1. Computer Graphics. 2. Macintosh (Computer) Programming. I. Robertson, J.D.
(James Douglas), 1943-. II. Title. III. Series.

T385.G747 1985 001.64'43 84-25254

ISBN 0-03-000477-2

Printed in the United States of America.

Published simultaneously in Canada.

5 6 7 039 9 8 7 6 5 4 3 2 1

CBS COLLEGE PUBLISHING

Holt, Rinehart and Winston

The Dryden Press

Saunders College Publishing

Dedicated to

Betsy and Celia

Their patience and encouragement reduce the burden that writing imposes on our private lives.

Table of Contents

	PREFACE	xiii
	INTRODUCTION	xv
	What are Computer Graphics? xv	
	Printer Graphics xvi	
	Character Graphics xvii	
	Pixel Graphics xvii	
	Macintosh Graphics xviii	
	Advanced Graphics xviii	
	MacPaint xix	
	Microsoft BASIC and Graphics Programming xix	
Chapter 1	SKETCHING	1
	The Brush 1	
	The Spray Can 5	
	The Brick Wall 7	
Chapter 2	MORE MACPAINT	11
	Marquee 11	
	Text 12	
	Application 1: Row of Macs 13	
	Application 2: Business Card 15	
	Application 3: Annotated Artwork 17	
Chapter 3	GOODIES	21
	Application 1: Venn Diagram 21	
	Application 2: HIPO Chart 26	

Chapter 4	ADVANCED DESIGN	29
	Application 1: Chelmsford, Waltham, Buenos Aires Fonts 29	
	Application 2: Icons 31	
	Application 3: Racing Invitation 33	
Chapter 5	A MACPAINT RECREATION	37
	Application: Tangrams with MacPaint 38	
	Using MacPaint to Produce Tangrams 40	
Chapter 6	PROGRAM PLANNING	45
	Top-down design 46	
	Steps in Program Planning 47	
	Structured Programming 52	
	Menus and Submenus, Main Programs and Subprograms 57	
	General Structure 60	
Chapter 7	MAC, THE USER, AND BASIC	63
	Introduction: In Defense of BASIC 63	
	Program Development Tools 65	
	CHAIN 66	
	COMMON 67	
	CLEAR to Increase Memory 68	
	User Interaction with the Mouse 70	
	Annotated Menu Listing 71	
Chapter 8	PIXEL GRAPHICS AND ICONS	77
	Application 1: Binomial distribution 79	
	Notes: 83	
	Application 2: Mathematically Derived Curves 84	
	Prolate Cycloid 84	
	Curtate Cycloid 86	
	Involute of Circle 86	
	Cardioid 88	
	Evolute of Ellipse 90	
	Hypocycloid of Four Cusps (Astroid) 91	
	Roses 93	
	Application 3: Birthdays 95	
	Application 4: Stars and Motion 97	
	Shooting Star 98	
	Enlarge Star program 103	
	Racing Stars program 105	
	Approaching Star program 108	
	Revolving Stars program 111	

Chapter 9	CLOCKS	117
	CALLs to Text Management Routines	117
	CALL TEXTFONT(n)	118
	CALL TEXTFACE(n)	118
	CALL TEXTSIZE(n)	119
	CALL TEXTMODE(n)	119
	Application 1: Wall Clock	119
	Application 2: Digital Clock	123
	Application 3: Two Clocks	125
	Application 4: Mantel Clock	127
	Application 5: News Room Clock	129
	Application 6: Egg Timer	131
	Conclusion	133
Chapter 10	THE LINE COMMAND	135
	LINE Instruction	135
	Using Angle and Radius with LINE	136
	Advanced Applications of LINE	138
	Tessellation	138
	ANGLEWALK: Random Tessellation with LINE	138
	Square Tessellation:	141
	Diamond Tessellation	142
	Four-pointed Star Tessellation	143
	Complex Tessellation	144
	Suggestions:	145
	Stars and Circles	146
	Sierpinski Patterns	149
	List of References on Fractals	152
	Centered Sierpinskis	153
	Bent Sierpinskis	156
Chapter 11	THE DRAW SUBROUTINE	159
	The DRAW Command's Syntax	159
	Motion Commands:	160
	Options:	162
	Modes:	162
	DRAW Subroutine	163
	Process Move	164
	Process Directed Move	164
	Process Pick Up Sign, Digits if any	165
	Applications of DRAW Subroutine	165
	Alphabet Generator	171

Chapter 12	MOUSE TANS	181
	Commonly Used Variables 190	
	Other variables 192	
Chapter 13	CHART APPLICATIONS	197
	Raw Data Program 197	
	Application 1: Piechart 199	
	Application 2: Icon Chart 204	
	Application 3: Bar Chart 207	
	INDEX	213

Preface

As every user knows, the Macintosh is a vastly different machine from the traditional personal computer. Its ancestry may include the venerable Apple-II, but it doesn't resemble it in any way.

Never before in the history of computing has a machine's usability been so dominated by its capacity to produce graphic images. The Macintosh is driven by its ability to generate excellent graphics. Even when it produces text and numbers on the screen, it does so by *drawing* them. It relies on its high resolution and superb built-in programs to generate every image that the user sees.

When the Macintosh was introduced early in 1984 (who can forget that Superbowl Sunday?) it had two applications programs that were user-ready— MacWrite, a friendly word processing program, and MacPaint, an incredibly different program to produce graphics. Within a month of the hardware's release date, Microsoft BASIC became available, further expanding the computer's flexibility. It was at this time that we became involved with Holt, Rinehart, and Winston Publishers to produce this book.

This book is the first in a series of books to be written for the Macintosh computer. It is certainly the most enjoyable project we have ever undertaken, given the fact that the Macintosh is so highly graphics oriented. It is not our first book on graphics for microcomputers, but it is certainly the most different of them. We have spent an enormous amount of time developing techniques and programs to produce graphics in the past several years, either for our college courses, or for a book we were writing. Only when we began this book, however, did we feel utterly comfortable with this engrossing topic.

Our goal is to stimulate our readers to explore the Macintosh's abilities. We provide ideas in the form of small sequences of activities for using MacPaint, or small programs in Microsoft BASIC. These activities must not be confused with the well-developed applications that we leave up to the reader to produce. Our aim is to give away some of the tricks that we have learned in our combined three dozen years of computing experience. We simply seed the territory. It's up to our readers to cultivate and harvest the rich rewards of computer-generated graphics applications.

All of the material in this book was prepared on an "as-delivered" Macintosh with 64K of ROM and 128K of RAM. It was delivered in March of 1984 in its standard configuration with a single 3.5 inch built-in disk drive. With the computer, keyboard, and mouse we also got a 9.5 inch carriage Imagewriter printer. The only software we used was the MacPaint (both the 1.0 and 1.3 versions, the latter becoming available in May of 1984) and the Microsoft BASIC. With this hardware and software, we explored the great graphics that the Macintosh can produce.

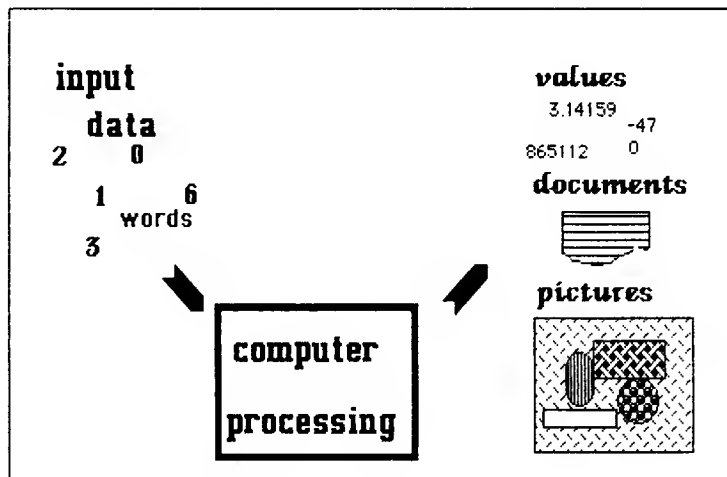
The book is in three major parts: Part I is an exploration of MacPaint graphics; Part II introduces good program design techniques; and Part III describes by example some of the possible graphics projects that you can design using Microsoft BASIC. It is in these last chapters that we pull out all stops and present some

graphics programs that could lead to full-blown commercial programs written by you, the reader. Our philosophy has always been to let the reader in on our development ideas. We begin playing around with techniques, and when we have a working program that uses some of the tricks we've discovered, we move on to something else. Our hope is that many of our readers will pick up where we've left off, and will produce well-designed, useful, rewarding commercial products.

INTRODUCTION

What are Computer Graphics

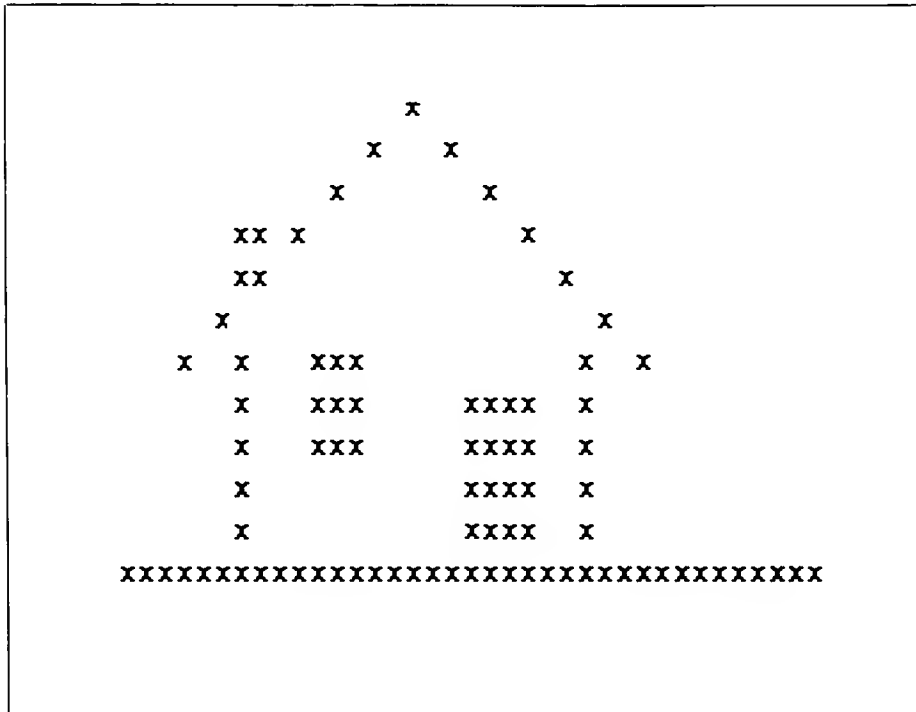
Simply stated, computer graphics is the most exciting output that a computer can produce. A computer — every run-of-the-mill computer — can deal with numbers and words. But it takes a major effort for these machines to generate pictures.



Pictorial output from a computer is the third of the three primary forms of computer output, and it is swiftly becoming the most popular. It is safe to say that the production of computer pictures is a relatively recent phenomenon. Computer graphics was too difficult and too expensive for general use until the 1970's. Before that time, computers were most often used in business to produce reports, and in science and engineering to calculate.

Printer Graphics

When a computer printer “draws” a picture of a puppy, or a spaceship, or any other image by printing a series of letters one line at a time, we refer to it as *printer graphics*. The sketch of the house below is an example of this kind of computer-generated graphic image.



This kind of computer-generated graphic image is simply a different use of the printer. Calculations don't enter into it, nor do special types of hardware or software.

Character Graphics

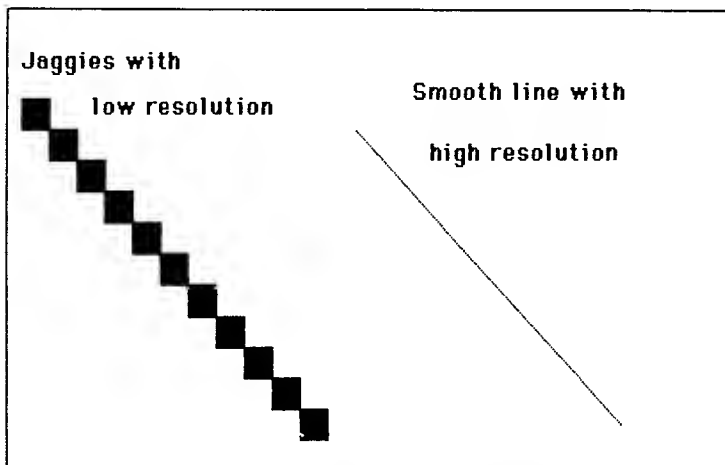
Another form of image that the computer can produce with only slightly more advanced hardware (and some tedious programming) is that sort produced using special characters aside from letters, digits, and punctuation. For example, some printer-computer combinations can make images using special characters like the ones below:

[] | / \ _ - + • △ ∞ ∩ □ ◇

Any image that is produced by printing these characters in straight lines one after another is simply an alternate form of the primitive printer graphics we mentioned above.

Pixel Graphics

It's only when the computer works together in concert with the screen and deal with individual spots of light that we can explore its full image-generating potential. All computer screen images are made up of individual spots of light or darkness that are called *pixels*, or *picture elements*. Each spot is represented within the computer as a single small piece of information, and several such pieces of



information can work as a group to produce a character. If you look closely at any character a computer forms when it prints, you can make out the pattern of dots that make up the whole.

Normally, a computer has a fixed set of patterns for upper and lower case letters, digits, and special characters. When it produces text or numbers for output, those patterns are projected onto the screen to make up the words and values. However, when such a computer works with graphics images, it assumes a different imaging process, whereby the entire screen is a map or a sketchpad onto which the graphic image is placed one dot at a time. This form of graphics is measured by the screen's *resolution*, or its number of rows and columns of individually addressable spots. Typically, these computers have a resolution of from 100 to 1000 columns and from 50 to 1000 rows.

If a computer has a resolution of 1000x1000, it must be capable of storing one million addresses for each image. This amount of memory is rarely available on personal computers, so most of these have a resolution of 350 to 500 rows by 200 to 400 columns. With this level of resolution, most images lose their jagged edges, or *jaggies*, so that circles look round and lines look straight.

Macintosh Graphics

The Apple Computer Company has made the Macintosh, patterned after its *Lisa*, to be a pure graphics machine. By this we mean that not only does it produce graphics images one small spot, or pixel, at a time, but it also produces its text the same way. There is no set way to print the letter "a" in the Macintosh, as is the case with most other computers. It is this feature above all others that distinguishes the Mac from its brethren. Because of this property, a screen full of text can have graphics painted onto it, and a graphic image can have words around it.

This ability to mix text and graphics is extremely powerful. It means that your drawings can be labeled easily. Also, it means that your traditional computer programs that generate text and values can be embellished with pictures produced by the computer. We explore both approaches in this book, as you will see.

Advanced Graphics

The topic of computer graphics is a highly technical one, and it is governed a great deal by what the computer in question can and cannot do. For example, the Macintosh cannot (as yet) produce color on the screen. Therefore we must consider color graphics images to be beyond the scope of this book.

We also avoid some of the more advanced topics in college-level computer graphics courses, such as windowing, clipping, animation, and three-dimensional

images. This doesn't mean that the Mac can't do these things; rather, the topics are of a technical nature that is beyond the scope of this particular book. The two topics we do cover in detail are MacPaint and Microsoft BASIC graphics programming.

MacPaint

The MacPaint program as supplied by Apple is such a superb piece of software that we cannot recommend it enough. MacPaint is a program with which the user can produce pictures *immediately*, and as an extra bonus can add text to those pictures. We devote the first five chapters to an exploration of this remarkable program.

Microsoft BASIC and Graphics Programming

BASIC is a familiar language to many. Secondary schools are adopting it more and more as a standard introductory programming language because it is so easy to learn and because BASIC programs can run on practically all small computers. Microsoft BASIC has become something of a standard among the varieties of this language, and the Microsoft BASIC that runs on the Macintosh is *extra* powerful because it has good graphics commands and because it takes advantage of the Macintosh set of pre-programmed graphics software.

The last five chapters of this book dig into the use of BASIC as a method for producing graphics. This is not to say that Microsoft BASIC on the Macintosh is limited to graphics. Instead, we are emphasizing the use of graphics programming as an enhancement to the large number of programs that produce text and values. We include several programs that produce business charts in this section of the book to show how graphics and text can mix and how graphics improves the computer output.

We have left many programs in their skeletal state for a purpose. We want you to take them and modify them to suit your needs. We expect you to push here and pull there, to tweak the programs to your liking. We invite your reactions to them, and would consider it a high compliment to see a variation of one of our efforts become a best-selling piece of software.

SKETCHING

This chapter starts the book with some of the elementary tricks you can use with that phenomenal piece of software, MacPaint™, written by Bill Atkinson of Apple Computer. If you still haven't invested in this application program, we recommend that you do so as quickly as you can, because it justifies anyone's purchase of the hardware. We will start with some primitive techniques. We can't guarantee that you will turn into an artist overnight. We didn't, as you will see. But we feel comfortable with the computer, and we have felt more free to "be artistic" as a result.

When you select and open MacPaint, either through the pulldown menus or by double-clicking, you are presented with your palette and sketchpad. Around the border you will find the patterns and techniques you can select; the mouse and your imagination are all that are necessary to do some pretty fancy sketches.



The Brush

The most direct tool for drawing freehand with the mouse is the *brush*, which is highlighted in Illustration 1.1. The squares in the left border show the tools that can be used. You can select any tool in the left border by clicking the mouse once. If you select the *pencil*, (found immediately to the right of the brush) as your tool, the shape your pencil will use is always a single thin line. As with the pencil, the shape that your brush will use is *not* the one shown at the bottom left corner of the palette, which is reserved for polygons, ovals, and freehand. To change the brush, you have to click the Edit menu item, (Illustration 1.2) then select the Edit Brush Shape as shown in Illustration 1.3.

In our initial play sessions with MacPaint, we discovered how forgiving the program can be when dealing with rank amateur artists. Consider the drawing of a tree. You have to draw the trunk, some large limbs, some smaller and smaller ones (more and more of them) until you are faced with all of the thousands of leaves. And if you want a foreground and horizon, you have those to contend with also.

To draw the trunk, select the fattest brush shape (from Edit Brush Shape), and make sure the pattern is pure black (the large square at the bottom left is black,

indicating the pattern selected). Then drag the mouse down, roughly in the middle of the screen as in Illustration 1.4. Select a thinner brush and now draw (with dragging) several large limbs (Illustration 1.5). You can go back to select an even thinner brush and draw some smaller limbs, but you don't have to (Illustration 1.6).

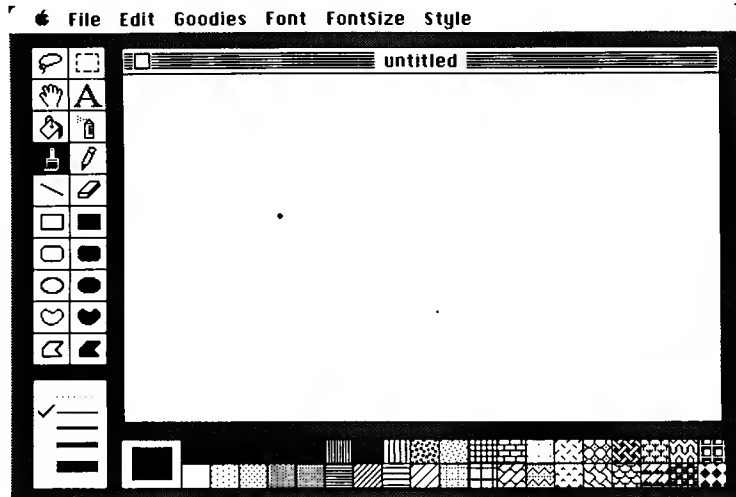


Illustration 1.1 The MacPaint palette and sketchpad

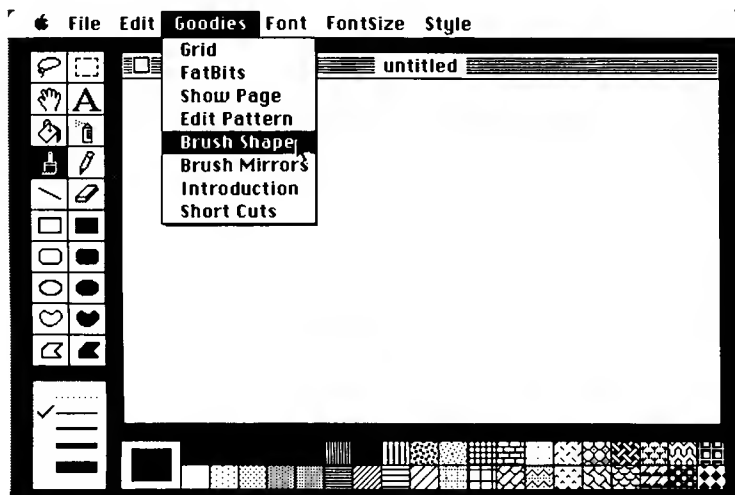


Illustration 1.2 Brush selection

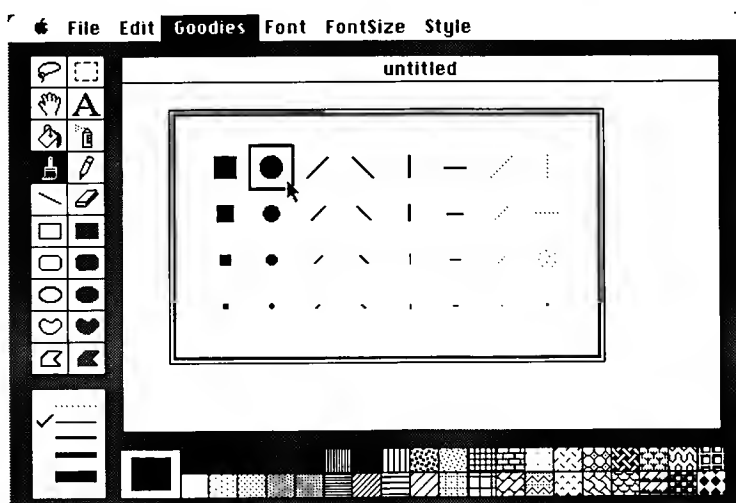


Illustration 1.3 Edit Brush Shape display

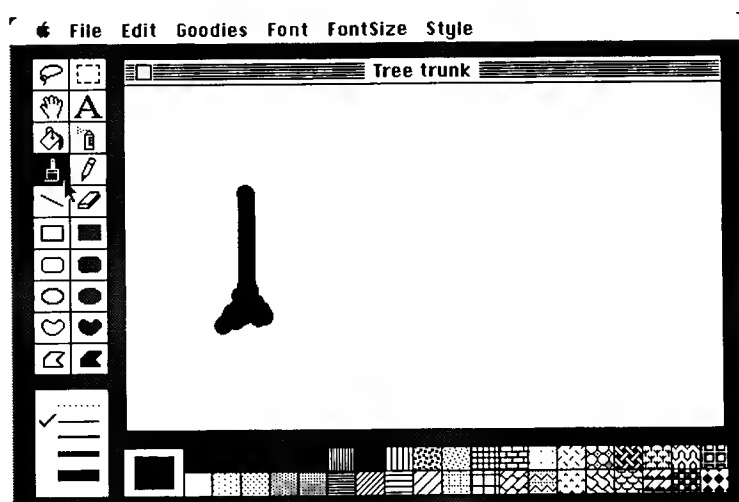


Illustration 1.4 The tree's trunk

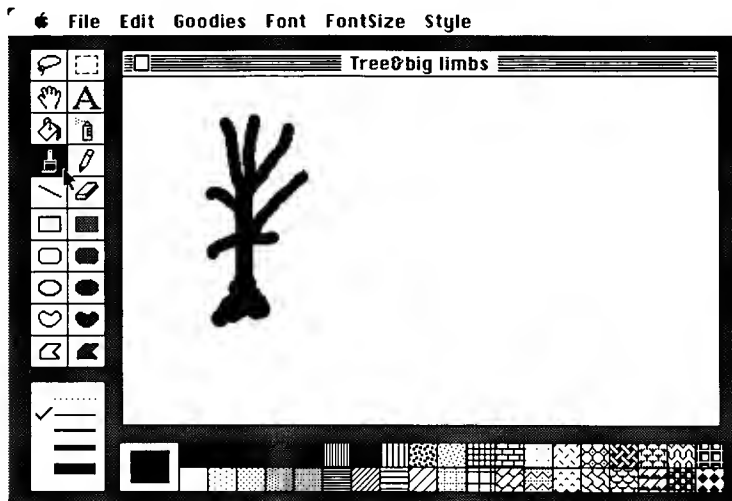


Illustration 1.5 The tree's major limbs

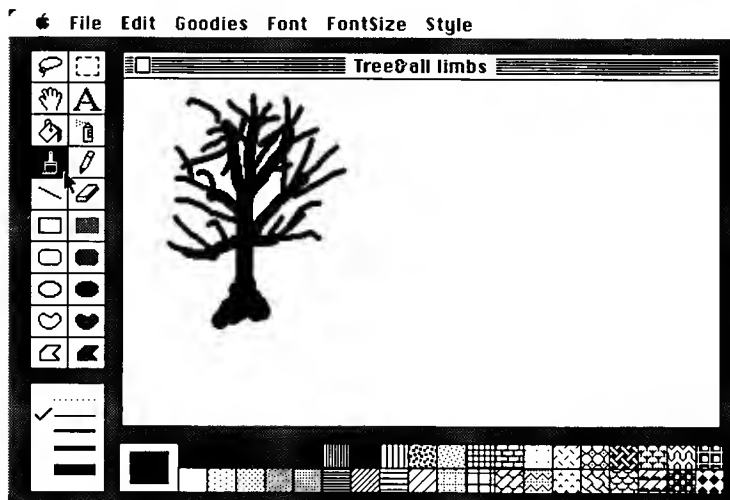


Illustration 1.6 All of the tree's limbs



The Spray Can

Now, here's what we mean by MacPaint being forgiving. Certainly you can't say that any one of the sketches is like a tree. The limbs are too uniformly sized, they end in that funny round shape, and there just doesn't seem to be enough of them. But now if you *spraypaint* some leaves, you can hide all of these imperfections, and those fat limbs are barely visible through the bushiness you've produced (Illustration 1.7). You spray on the leaves by selecting the spraycan from the tools along the left side and dragging rather quickly through the branches of the tree. *Hint:* If you see that the tree is turning black (you left the spraycan on too long in one place) you can lighten it by selecting a different pattern — say, the random dots in the top center, or the pure white pattern at the bottom left — and spray that pattern.

Now, you need to draw your horizon and some background and foreground.

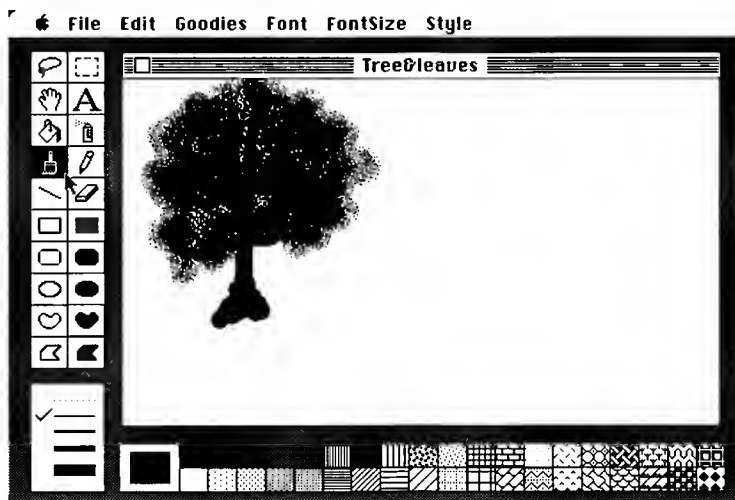


Illustration 1.7 Tree after spraying leaves

Using a brush of medium thickness and a dark (but not black) pattern (such as the fourth one in the top row), drag a line across the screen. That light streak across the trunk of the tree can be repaired later with a black brush. Then pick up progressively lighter patterns and spray the area below the horizon from top to bottom (Illustration 1.8).

We will leave you to your own devices in polishing this masterpiece. You need to take care of that light streak in the trunk, and to shape the bottom of the trunk with the brush so that it doesn't look so squatty. Also, you can spray some fluffy clouds in the sky, or a moon, or birds, or a Mack truck if that's your preference (Illustration 1.9).

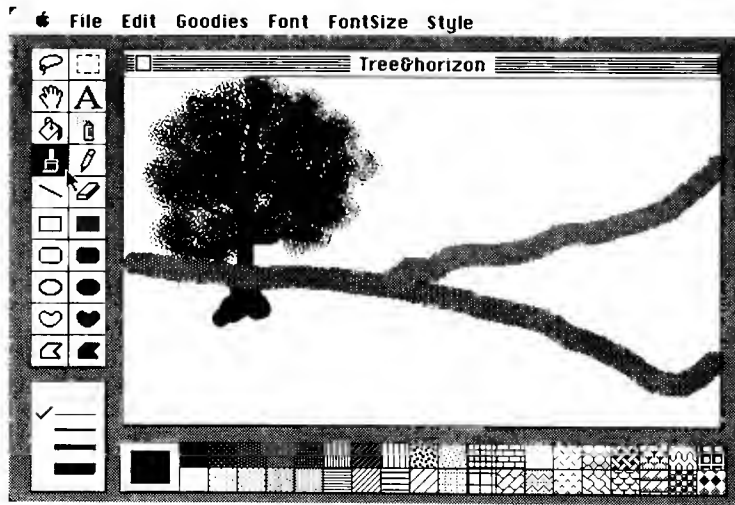


Illustration 1.8 Tree with horizon

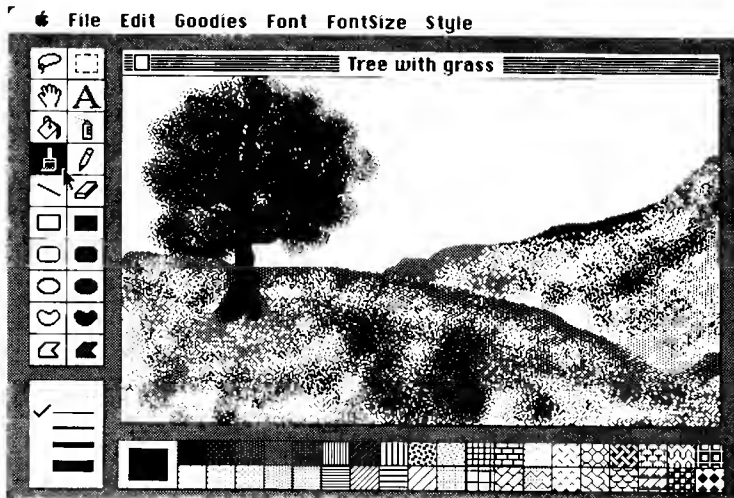


Illustration 1.9 Tree with grass



The Brick Wall

Some of the most enjoyable aspects of MacPaint are the pre-programmed patterns that you have at the base of your palette. You have fish scales, webbing, regular dots, random dots, diamonds, waffle tiling, bricks across and diagonal, even ceramic roof tiling for those houses in the Provence area of Southern France. One of our favorite applications of MacPaint is the Graffiti Wall. This sketch is simply a wall of bricks onto which we spray, or crayon, or paint, our favorite sayings as in Illustration 1.10.

Try this:

1. Select the empty rectangle from the tools (sixth one down, leftmost column).
2. Place the cursor (mouse arrow) in the upper left area of the sketchpad.
3. Drag the mouse to the lower right corner of the sketchpad, and let go.
4. Fill this rectangle with bricks. Select the horizontal brick pattern.
5. Select the *paint bucket* to completely fill the rectangle with the brick pattern.
6. Click the mouse anywhere within the rectangle. Voila! Your brick wall.
7. Select a brush shape to your liking and write your message.
8. To put a crack in the wall, select a white pattern and brush in the crack.

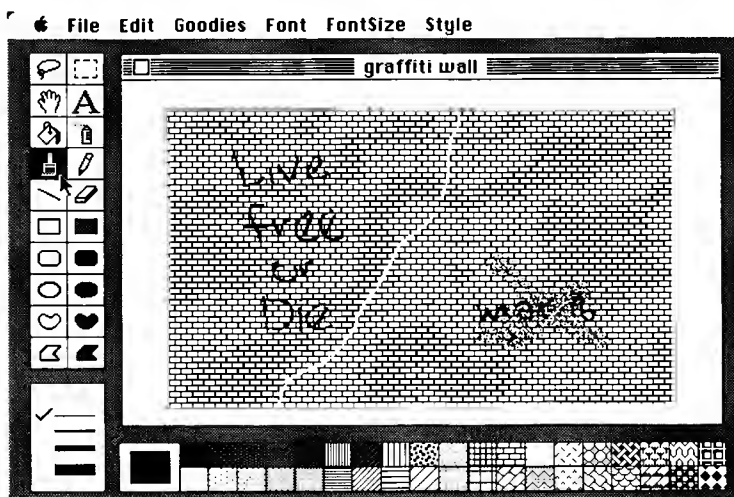


Illustration 1.10 The graffiti wall

9. To erase a message you've already written, *don't* use the eraser! You will erase your bricks as well. Select the brick pattern and the spraycan, and spray the bricks back on. Or you can paint the bricks back on using the brush. As you brush or spray, that pattern appears as a replacement of the old.

We will end this chapter with a few selections from our own ever-growing scrapbook of art. All of the following sketches (Illustrations 1.11 through 1.14) were done with brush, bucket, spraycan, rectangle, and circle. We used some of the existing patterns, and we used the *Edit Brush Shapes* feature.

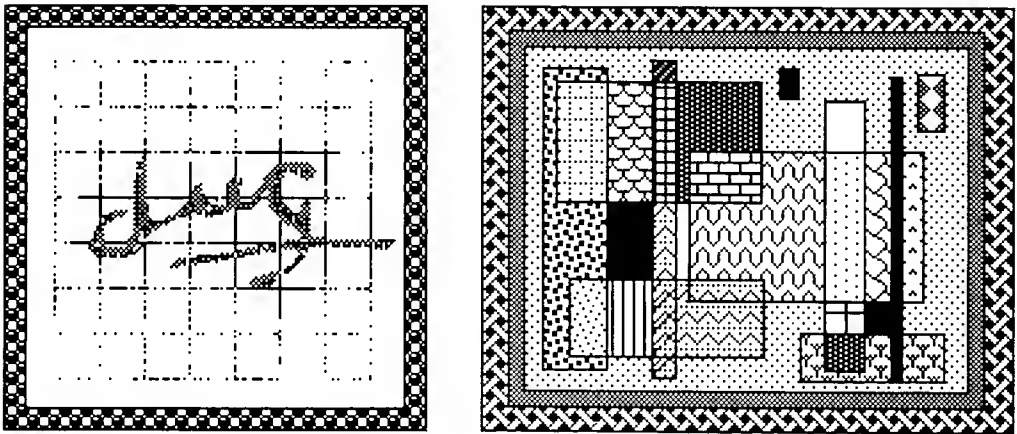


Illustration 1.11

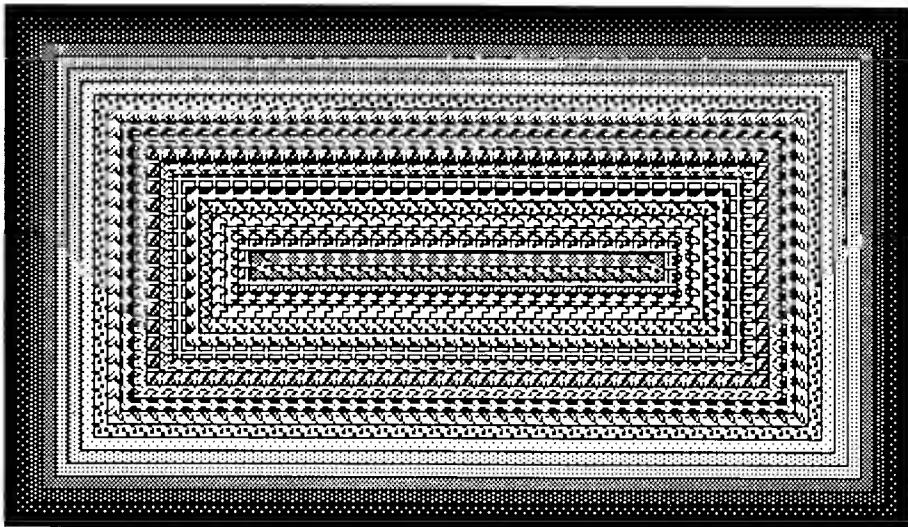
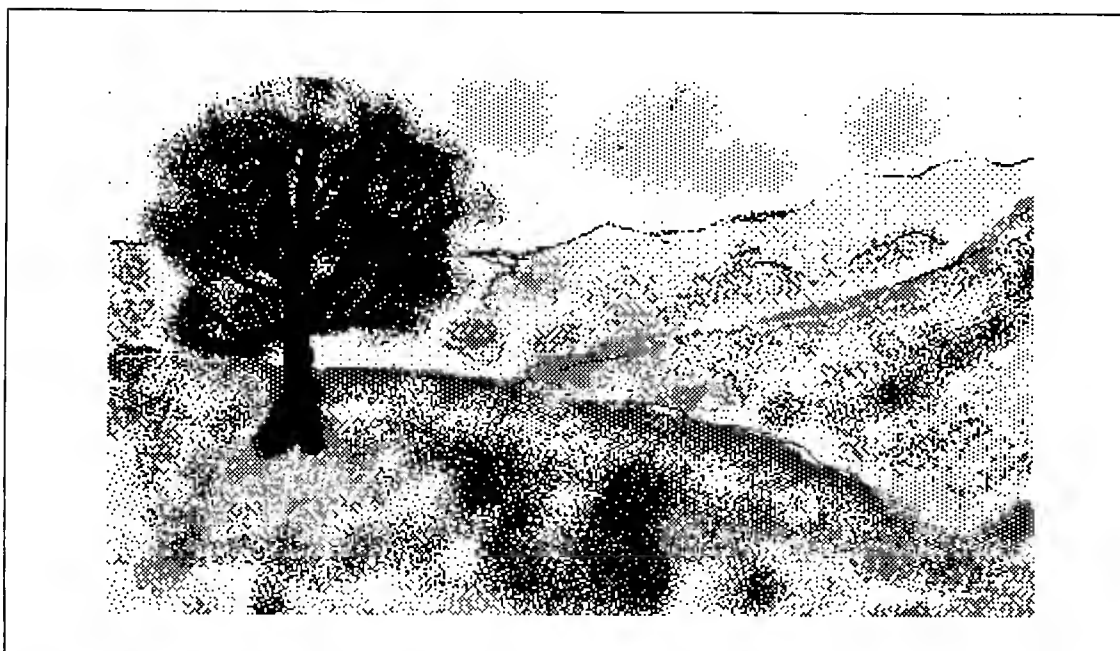
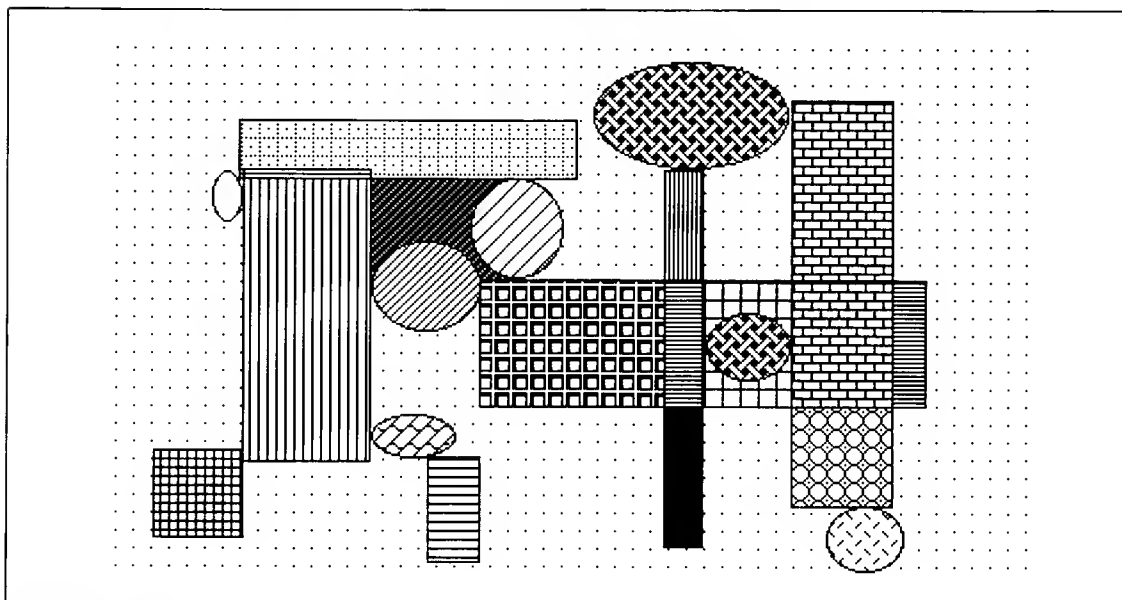


Illustration 1.12

**Illustration 1.13****Illustration 1.14**

One last technique that you will want to use often, or until you run out of disk space, is the save activity. Simply click the *file* menu entry, drag to the save entry, and follow the directions.

C H A P T E R T W O

MORE MACPAINT

The first chapter introduced the MacPaint sketching area, and demonstrated a few of the tools and patterns that you have available to you. This chapter expands on those features, and puts particular emphasis on the use of two features that make MacPaint especially powerful — the marquee and the alphabet.



Marquee

The *marquee* is the top tool in the second column, just above the capital A. Notice that it is made up of dashed lines, unlike either the empty rectangle or the filled rectangle in the sixth row. This signifies that if you select the symbol and click-drag a rectangle on the screen, you will draw not the usual solid-line rectangle, but an area surrounded by a series of short lines. The lines rotate around the area like a movie marquee, hence the name for this tool.

Once you have selected the marquee and have used it to outline an area on the screen, you can do several things:

1. **MOVE:** Place the pointer inside the rectangle and drag the picture to a new area of the screen.
2. **COPY:** Hold down the **OPTION** key and drag the picture to another area. The original picture stays where it was, and the copy, if it was dragged only partly off the original, will overlay it.
3. **SHRINK:** Hold down the **COMMAND** key and drag . You can stretch or shrink in either direction, vertically or horizontally. This technique can modify a text font or change the aspect of a figure.

Application 1: Row of Macs

We have chosen this application first because it demonstrates both the marquee and the alphabet, or text generator. The screen image shown in Illustration 2.2, which we have left untitled, shows a design that is familiar to all Mac users. The stylized outline of computer, keyboard, mouse, and apple are on the Scrapbook, so we didn't have to sketch that image. Here's the list of steps you need to perform in order to produce the image in Illustration 2.3.

1. Select Scrapbook from Apple menu (at top left of screen).
2. Select Edit menu and COPY scrapbook selection to clipboard. You see nothing to indicate the action, but after this activity, you have the image both in the scrapbook and on the clipboard. Be careful: do not CUT the image from the scrapbook, because that will remove it from the scrapbook, and it will not be available to you for other purposes.
3. CLOSE the scrapbook — click the square at the top left of the scrapbook window.
4. Select PASTE from the Edit menu. By selecting the activity, you place the contents of the clipboard into the MacPaint sketch area. Notice that the image seems to vibrate. It is made up of blinking dashes. And if you move the mouse, the cursor (pointer) on the screen is no longer an arrow but a lasso. This is like a marquee, except that its shape conforms to the image's outline, and not a free-floating rectangle outside the image.
5. Place the lasso inside the vibrating figure until it is a vibrating arrow. Take care with this operation. You can't operate on this image if the pointer is a

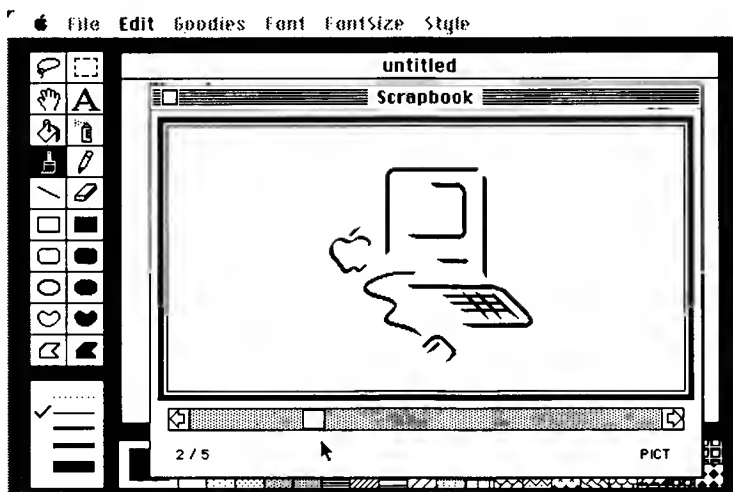


Illustration 2.2 The Scrapbook, source of our design

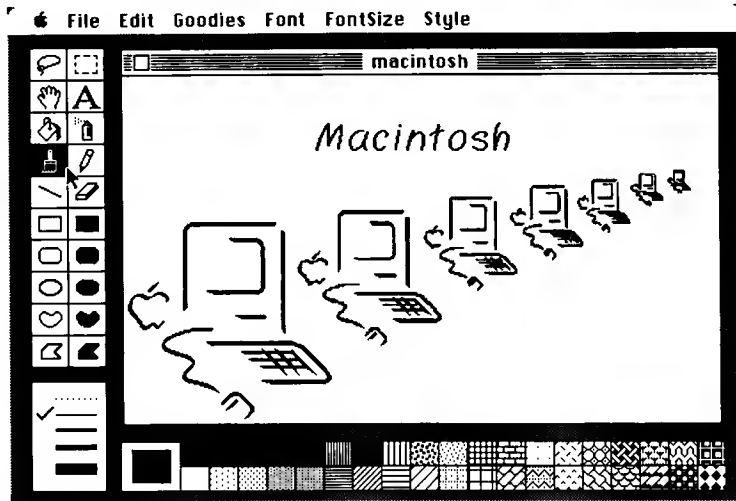


Illustration 2.3 Row of Macs

- lasso. It must be a vibrating arrow. If you try to click-drag the image elsewhere on the screen while the pointer is a lasso, the image suddenly gels and you find your pointer forming a line, the beginning of a new lasso.
6. Click-drag the vibrating image to the bottom left of the screen.
 7. Select the Marquee and form a smaller square slightly above and to the right of the now-solidified image. This step is one of the Mac's fascinating features. What you have done is to provide the PASTE command in the Edit menu with a smaller window to use for placement of the Clipboard's image.
 8. PASTE the image into this smaller marquee.
 9. Repeat Steps 7 and 8 until you are satisfied.
 10. Select the Los Angeles font from the Font menu.
 11. Select font size 24 from the Style menu.
 12. Select A—text from tools (just below marquee).
 13. Move the cursor (now a bar) to top of screen, and click it into position. If you don't click the cursor, you cannot type text. When you click the cursor, you can tell that you are ready to type because the cursor is now a blinking bar.
 14. Type "Macintosh".

An alternate procedure is a bit more tricky, but will give you practice in handling the SHRINK operation with the marquee. Replace Steps 7 and 8 above with the three steps shown below:

- a. Select the marquee and frame the full-size image on the screen. Do this by clicking at the upper left corner of an imaginary rectangle around the image, then dragging to the bottom right of that imaginary rectangle. You form a rectangular marquee around the image.
- b. Drag a copy of this image (hold down the Option key while you move it to its new position, slightly higher and to the right of the old image).
- c. SHRINK this new image to size. Click the arrow at the bottom right of the marquee, and drag it slowly toward the upper left corner.

Application 2: Business Card

The principal reason we have chosen to discuss this application here is to introduce the concept of constraint, which allows you to sketch straight lines vertically or horizontally, draw perfect squares or rectangles, and even erase more neatly. We also use the lasso to select and move areas of design or text (Illustration 2.4).

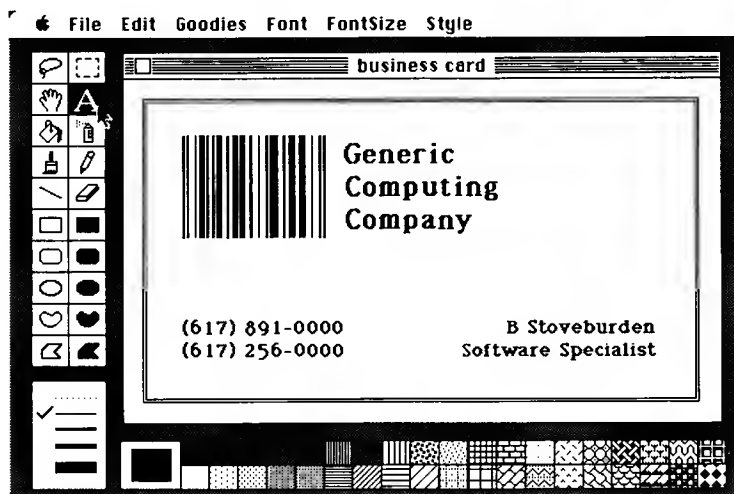


Illustration 2.4 Business card

1. Draw the frame of the business card. There are several ways you can do this. The simplest procedure is to select the thickness of the line from the menu at the bottom left of the palette, select the rectangle, and click-drag the business card's outline. You can repeat the procedure inwards for a frame of several lines.

Or you can have a fancy frame made up of one of the patterns at the palette by selecting a brush shape, palette pattern, and brush. Then draw the edges, but hold down the shift key (constrain your motion to a straight vertical or horizontal while dragging to keep the lines straight. The hard part of this procedure is the cleaning up afterwards, when you have to erase the extra overflow at every corner — unless you're lots better than we are at stopping where you should. *Hint:* Select the Grid from the Goodies menu to keep your brush from drawing too long a line.

2. Select line (fifth down in leftmost column) to build the UPC symbol. Use constraint again to keep your lines vertical, and select your line thickness from the bottom left menu. Don't worry about the length of the lines, as you will clean up the top and bottom later.
3. Use the Eraser to clean up the UPC symbol. Use constraint (the shift key) to keep the eraser moving in a horizontal line.
4. Select New York font, fontsize 18, style bold. To select the style, you can pull down the menu and click the selection, or you can type *command-B* (hold down the command key and press B).
5. Select the alphabet and click pointer (now a vertical bar) to the right of the UPC symbol. Type:

Generic
Computing
Company

6. Lasso the letters and drag them into their correct position.
7. Select Fontsize 12, A. Type (in vacant location at bottom left):

(617) 891-0000
(617) 256-0000

8. Lasso and drag phone numbers into position.

9. Click in lower right corner of screen. Select *Style = Align Right* (Illustration 2.5) and type:

B Stoveburden
Software Specialist

10. Lasso and drag this latest entry to appropriate spot on card.

We encourage you to design your own business card, or greeting card. Imagine the possibilities. With Cairo font full of little designs and your own artistic skill and imagination, you can design some pretty fancy documents — whether they are invitations, greeting cards, business cards, posters, or announcements.

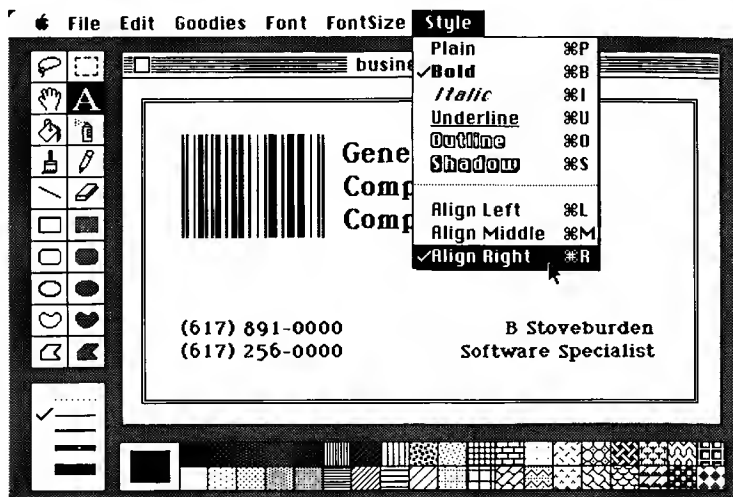


Illustration 2.5 Style menu with selections highlighted

Application 3: Annotated Artwork

Illustration 2.6 is a simple sketch of two heads facing each other, but its production requires the use of some advanced techniques. You will use constraint to make a perfect circle, and you will flip an image.

1. Using the filled circle with the standard black pattern, draw the head by dragging with constraint (shift key) to form a circle.

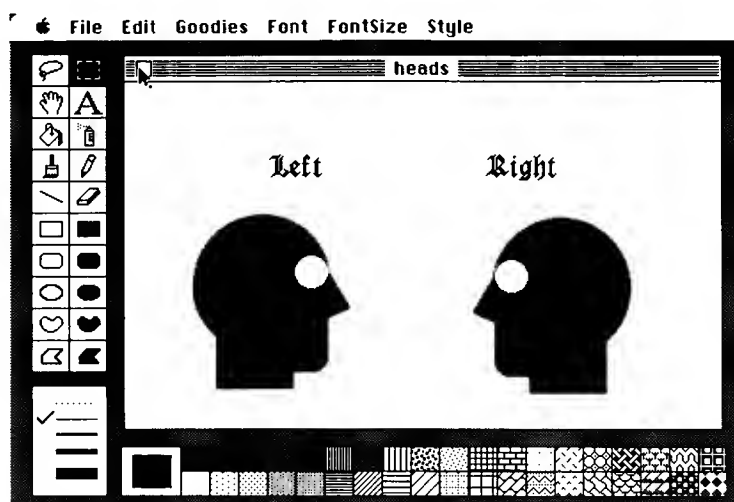


Illustration 2.6 Two heads facing each other

2. Select the white pattern and draw a small white circle (drag with constraint) in an open area of the screen.
3. Lasso the small white circle and drag it into position as the eye.
4. Select the Line and draw from eye to tip of nose, then toward face, then down the length of the chin. On this last line, you can use constraint or you can take care to draw a perfectly vertical line. Then draw a horizontal line toward the neck, another line down for the neck, another horizontal left, and finally the vertical line at the back of the head. When you are doing this sketching, remember that all of these lines will be used as outlines to be painted black, so it's safe to go too far toward the center.
5. Fill in the blank spaces with the paint bucket and the black pattern.
6. Select Fatbits from the Goodies menu to touch up the chin so that it is more rounded. Notice that the upper left of the sketchpad has a miniature picture of the enlarged (fat bits) section on the screen. This little sketch is often helpful for you to locate where you are on the original drawing. Select the *grabber* to move the image around so that the sketchpad is magnifying the chin. Round the chin by using the Pencil to erase individual pixels. A *pixel* (from Picture Element, is the smallest graphic element available on a computer.) Every click on a black pixel erases it, and every click in a white area draws a black pixel.
7. Select the Marquee and outline the entire head. Duplicate it by dragging it with the option key pressed. Be sure to drag it off the original sketch completely. Then, while the marquee is still around the duplicate, select Flip horizontal from the Edit menu.

8. Select Alphabet, Fontsize 18, London Font, Style Align middle. Position the cursor above the head on the left, type "Left", some spaces, and "Right".

This exercise may seem trivial, but it contains the elements of many applications beyond the simple sketch we have provided. Consider, for example, stretching or shrinking the head in either direction to change the aspect of the head. Or you could shrink one or both of the heads to token size so that you can use them as symbols within text.

The power of the *flip horizontal*, *flip vertical*, and *rotate* in the Edit menu must not be overlooked. With these commands, for example, you can produce a fancy frame corner once and position it at all four corners of your sketch.

GOODIES

Two examples of outstanding graphics features are Fatbits (which you have already seen) and Grid in the Goodies menu. Here we have two tools that are not unique to graphics systems, but which have been developed with ease of use in mind. The two applications that we will show you in this chapter rely heavily on these Goodies, and we encourage you to rebuild them along with our descriptions. And as you do so, consider the wide breadth of applications that you can develop on your own.

The Venn diagram (Illustration 3.1) illustrates yet another use of mixed text and graphics, but in this case the layout seems neater, more uniform. Look at the seven rectangles in the illustration; they are used as keys to the seven patterns that the three overlapping circles have enclosed. All are exactly the same size. If you take a ruler to the circles, you will see that first, they are indeed circles and not ellipses that are almost circles, and second, all circles are exactly the same size. These

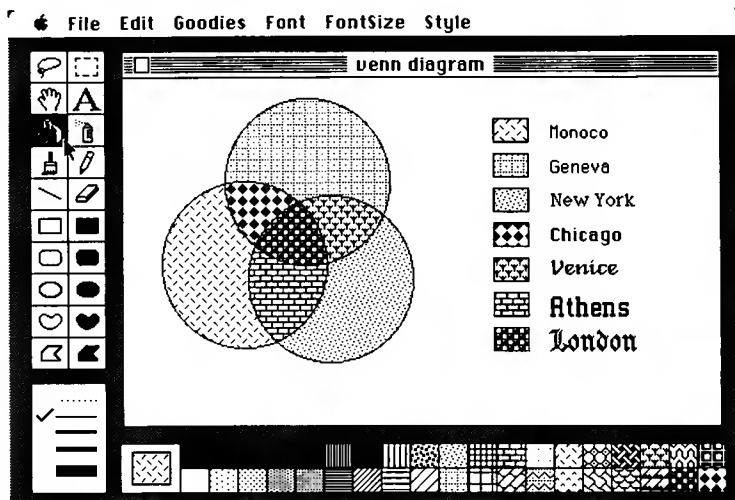


Illustration 3.1 Venn diagram

properties were derived with the use of constraint, Grid, and Fatbits. In order to construct the Venn diagram:

1. Select Circle from the available tools. Holding the shiftkey down for constraint, drag diagonally to form a perfect circle.
2. Make a hole in the side of the circle. To show the purpose of the hole, let's go through a small exercise. You know that you want to have three overlapping circles, and that you want to see all circles completely. If you make two copies of the first circle, you can drag them into their approximately correct positions, but you won't see them in their entirety. Do this:
 - a. Lasso the circle you have drawn. Holding down the Option key, drag a copy of the circle to the appropriate place. Notice that you can't see that part of the original circle which is under the copy.
 - b. Repeat the step above, making a third circle. This one will seem to rest on top of the other two. These aren't transparent circles; rather, they are solid disks! Not what we want at all.
 - c. Double-click the eraser to get rid of everything, because it will be much simpler to start fresh.
 - d. Now let's do it right. With a circle on the screen from Step 1, double-click the pencil to get to Fatbits.

The pencil is selected automatically when you get there. Click the pencil on any dot of the circle so as to leave an opening. Get out of Fatbits by clicking the miniature picture at the top left of the sketch-pad.

3. Duplicate the circle twice. Do this by using the lasso to make the circle into its own Marquee. You select the lasso, trace a rope around the circle, and let go. The circle will seem to vibrate, just as the Macintosh symbol did in the previous chapter when you pasted it into the sketchpad from the clipboard. Move the lasso toward the circle until it turns into an arrow, and then drag it to the location for the new circle. Notice that this time, you see the edges of the circle below, which are necessary to outline the intersecting areas.
4. Patch the three circles where you had placed the hole to allow the leak. Now you don't want it. Get into Fatbits by double-clicking the pencil (or select it from the Goodies menu), and find the three places where you had created the holes. If you can't see a hole on the portion of the screen that is displayed, select the Grabber and using a drag operation on the screen with the Grabber, locate each hole. To patch, pencil in a new dot, or pixel, at each hole.
5. Select a pattern of your choice.
6. Select the paint bucket from the tools.
7. Click the pattern into one of the seven areas outlined by the three circles. If your circles have not been closed properly, the paint will leak from one area to another. In that case, *don't click*. Select UNDO from the Edit menu. This will remove the last operation to the previous click, in effect removing the paint where you don't want it.
8. Repeat Steps 6 and 7 six more times, once for each open area of the three-circle pattern.
9. Select Grid from the Goodies menu. This doesn't seem to change anything. In effect what has happened is that an invisible grid has been placed in the sketch area that restricts alphabetic characters, lines, and shapes to follow those lines of the grid.
10. Select the Rectangle tool from the leftmost column. Drag seven rectangles in a column, one at a time. You will notice how much help the grid is because it restricts the rectangle's size.
11. Select each of seven patterns one at a time, and using the paint bucket, fill each of the seven little rectangles with a different pattern to conform to the seven patterns in the three circles.
12. Select the Alphabet as a tool. Then for each of the seven fonts available select font size 12 and enter the font name next to the rectangle. Again, the invisible grid helps you to align the text.

This application was a natural tease to a second version (Illustration 3.2). We selected three classes of people and made them overlap, as does the Venn diagram on the previous page. We learned a good deal about how to make a full-page-size diagram in the process, so we'd like to share our findings with you.

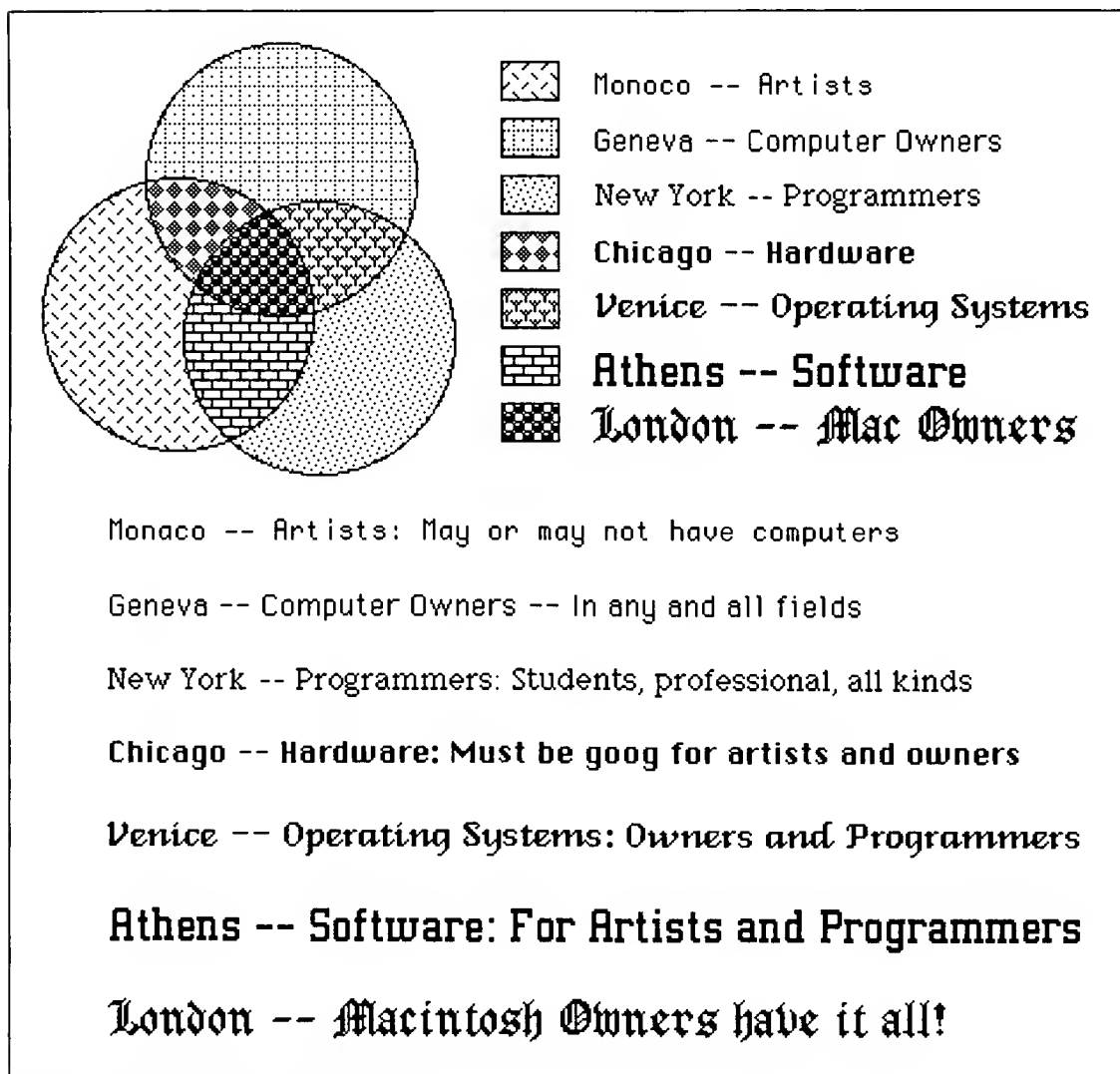


Illustration 3.2 Artist's Venn diagram

Follow these steps to duplicate Illustration 3.2:

1. Establish where the original sketch is on the entire page. Select Show page from the Goodies menu, and you will see where the sketch is by the position of the dashed rectangle. If you wish to move it elsewhere on the page, drag this rectangle there. Then return to the sketch itself by clicking Ok on the screen (Illustration 3.3).

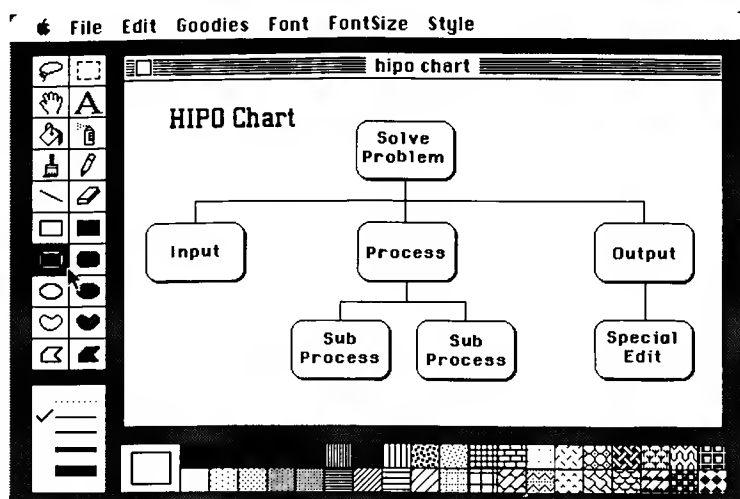


Illustration 3.4 HIPO chart

Application 2: HIPO Chart

The power of MacPaint as a text management tool becomes evident here. The HIPO Chart, Hierarchy of Input-Process-Output, (Illustration 3.4) is a commonly used documentation tool when large, complex programs or systems of programs are written. Here we show a sample portion of a typical chart, not to describe an existing system, but to show the process of making the chart.

1. Select the all-black pattern from the palette.
2. Select the filled, rounded rectangle (seventh down, second column) as a tool. Drag to create an all-black rounded rectangle.
3. Select the all-white pattern from the palette.
4. Position the pointer just above and to the left of the upper left corner of the drawn black rectangle. Drag a white rectangle across and down, until it almost covers the old black one. If you go too far and let go of the mouse button, all is not lost. Simply select UNDO from the Edit menu and try again.
5. Lasso this shape and drag it to the top-center of the screen.
6. Holding down the shift key to constrain movement and the Option key to leave the original behind, drag a copy of the rectangle straight down to the center of the screen.
7. Release the mouse, then press it again to select the new copy of the rectangle. Drag it to the left with Shift and Option keys pressed.

8. Release the mouse, then press it again to select the newest copy of the rectangle. Drag it all the way to the right of the screen with shift and option keys pressed.
9. Release, select the newest rectangle shape, drag (shift-option again) straight down to the lower right of the screen.
10. Release, select the new copy, drag (with shift-option) straight left, and position just right of center below the center rectangle.
11. Release, select the new copy, drag (shift-option) straight left again, to complete the placement of the seven identical rectangles.
12. Select Athens = Font, Fontsize 18, Alphabet (A), and type:

HIPO Chart

13. Select Seattle = Font, Fontsize 10, Align middle, Bold. Click to middle of each rounded rectangle and type labels.
14. Select Line as a tool.
15. Draw lines connecting rounded rectangles. Constraint is not necessary, though it could be useful.
16. Use the eraser to touch up any overdrawn lines.

Consider the possibilities: Program flowcharts, system flowcharts, organization charts, Gantt charts, PERT charts, ... the list of uses for this simple series of techniques is endless. We leave you to their exploration and exploitation.

ADVANCED DESIGN

This chapter deals with two subjects only—and both relate closely to the use of Fatbits in the Goodies menu. The two topics are the production of a font and the production of *icons*. You are familiar with this term already, because your Macintosh uses icons to symbolize files, pictures, applications, and various other types of operations or software. An icon is simply a graphic image that acts as a symbol. When that icon is used properly, it becomes associated with a specific action, or property, or some other characteristic. We will introduce you to the production of icons that relate to Olympic sports activities, although you could have similar icons for many other areas of interest.

Application 1: Chelmsford, Waltham, Buenos Aires Fonts

The font we show in Illustration 4.3 may seem familiar, sort of like a weird variation of the Chicago font that the Mac uses in its top-line menu displays. Upon closer inspection, though, you will spot several differences, which of course make it a font of its own. For example, the capital letter “S” is quite different, with fat serifs at the tails, instead of the thin ones Chicago uses.

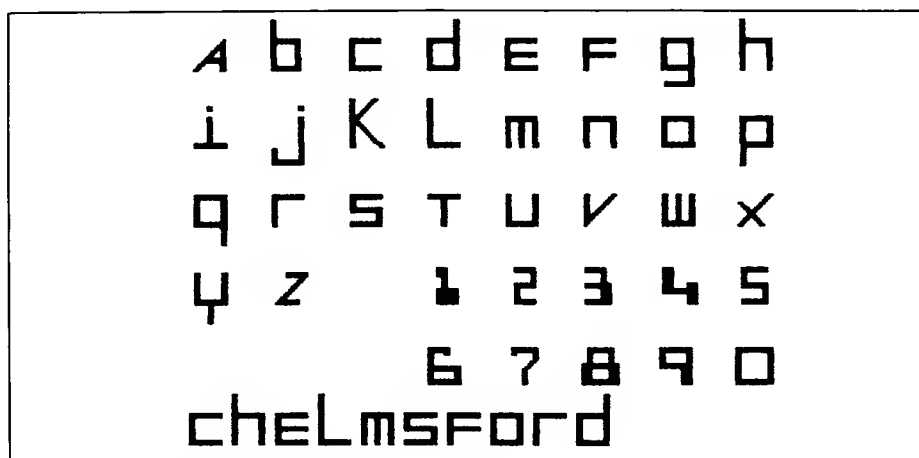


Illustration 4.1 Chelmsford font

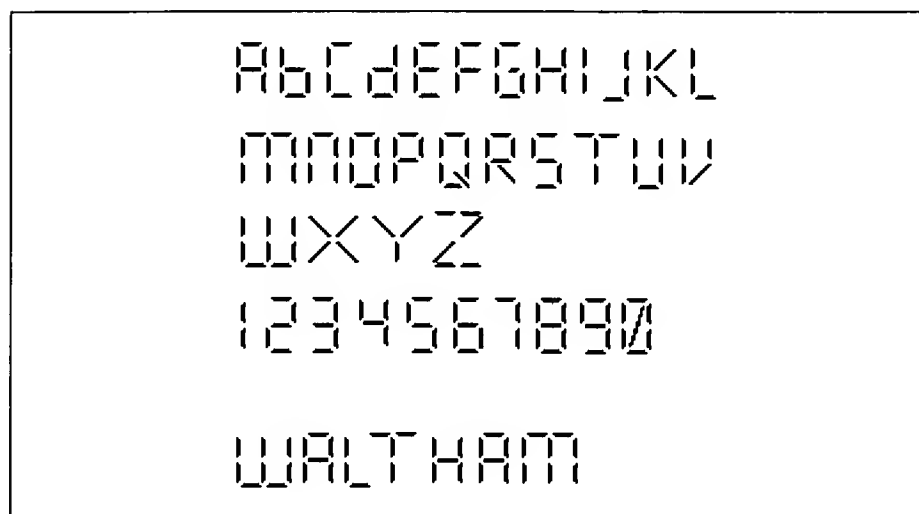


Illustration 4.2 Waltham font

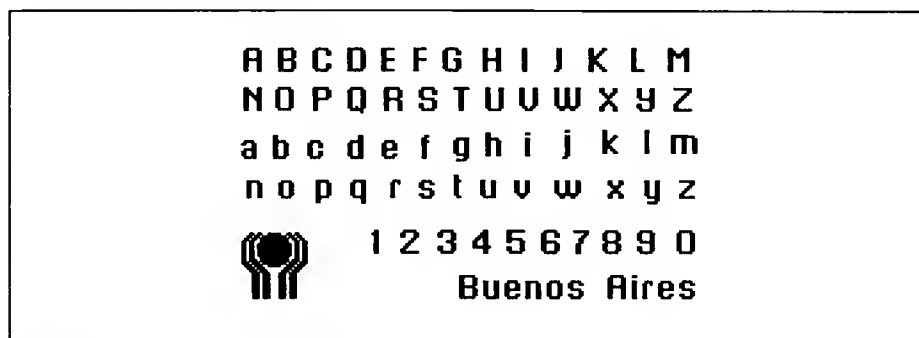


Illustration 4.3 Buenos Aires font

How did these letters get to be formed? There are *three* ways to produce fonts easily on the Macintosh using MacPaint:

1. Produce a screen image of all letters (upper and lower case) and digits, and if you are brave, all special characters as well, of an existing MacPaint font—such as Geneva, or Chicago, or Venice. Then isolate each letter and alter its pixel image in Fatbits to suit a particular style of your own. When you are satisfied with your font, save it as a MacPaint file.
2. Start from scratch, preferably in Fatbits, by designing your own font. This is especially difficult, as it requires a particular talent in producing images of letters whose overall style is consistent from one letter to the other, and in upper and lower case. You should try it once, if only to recognize the graphics designer's peculiar skills.
3. Go to the library, find the section of Graphics or Graphic Arts or Printing and thumb through the books and magazines until you find some with blown-up images of fonts. They exist, and in some libraries they are plentiful. When you find a font to your liking, transfer the letters dot-by-dot from the source to the Macintosh in Fatbits. This is tedious and unoriginal, but very effective.

Application 2: Icons

Two types of icons are shown in Illustration 4.4: The square images are symbolic of several Olympic sports and vehicle types. You can easily identify track, swimming, hockey, bicycling, skating, and shooting. You can also spot a car, a van, a taxi, and a car with key, perhaps indicating a safe or secure parking area. One square shows a diagonal bar from upper left to lower right. That icon is understandable in any language as "NO".

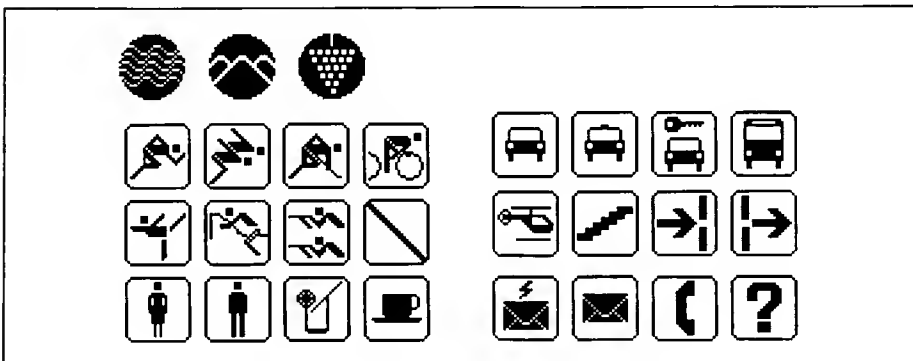


Illustration 4.4

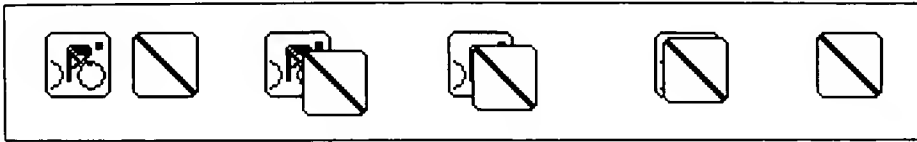


Illustration 4.5 Bicycle icon covered by “NO” icon.

How would you produce a combination icon, say the bicycle and “NO”, for a “NO BIKING” sign? If you simply lasso the “NO” and drag it over the bicycle icon, the result is the “NO” only, with the bicycle icon hidden beneath it as in Illustration 4.5.

What you must do is to open the “NO” icon so that its image does not hide the bicycle icon when you drag it. As you did in the previous chapter, you must provide a flaw in the icon’s outline to allow it to *bleed* over the image. Also, we recommend that you proceed carefully, *copying* the icons when you use them so that you can always keep the original handy. Do this:

1. Lasso the bicycle icon. Drag with the Option key down, into an open area of the screen.
2. Lasso the “NO” icon. Drag a copy (Option key down) into the open screen area.
3. Get into Fatbits to make an opening *on both sides* of the “NO” icon. You must open both sides because you have two closed areas. That diagonal in the center would prevent bleeding from one area to the other—unless you make a flaw in it, of course.
4. Lasso the “NO” icon and drag it over the “BIKE” icon until the edges coincide exactly. When you let go of the mouse, the result is a “No Biking” icon as in Illustration 4.6.

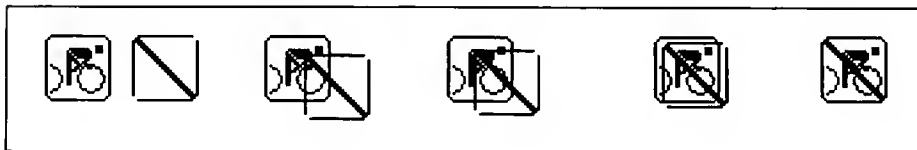


Illustration 4.6 “No Biking” icon properly done

The circular icons also shown in Illustration 4.4 are quite different, but their production is not any more difficult. Consider them as filled circles of uniform size, each with their symbolic content in white—a simple exercise in Fatbits.

Text may be placed next to an icon (Illustration 4.7) by dragging the appropriate letters from either your own font or from some letters you have produced in a Macintosh font of your choice. In either case, it’s best to use the Grid feature in the



Illustration 4.7 Estadio del Mar icon in Fatbits

Goodies menu to position the letters. Be careful about Grid, though, because it controls vertical as well as horizontal motion. You might prefer to use your own eye-hand coordination to position the letters free-hand.

Application 3: Racing Invitation

The last application in this chapter is a simple idea, yet could be an excellent application for the Macintosh. Consider the many public functions that organizations sponsor and want to advertise. The production of posters, flyers, advertising folders, publicity notices, even the tickets themselves to various events, are all appropriate products for you and the Macintosh. In Illustration 4.8 we present a possible invitation to a race. It is made as a folding sheet with five separate pages. We have illustrated four of its ten surfaces. One contains the title of the event, the second a pattern of icons symbolizing the road race, and the last two the ticket number.

The image was produced using several of the tricks you have learned. First, the track icon can be duplicated and the two positioned next to each other. Then the pair is duplicated as a set of four, that set doubled, and so on until you have the desired pattern. The entire page of icons can be cleaned up in Fatbits so that joint bars between the rows and columns of icons are uniform.

The invitation's text is 14-point Venice.

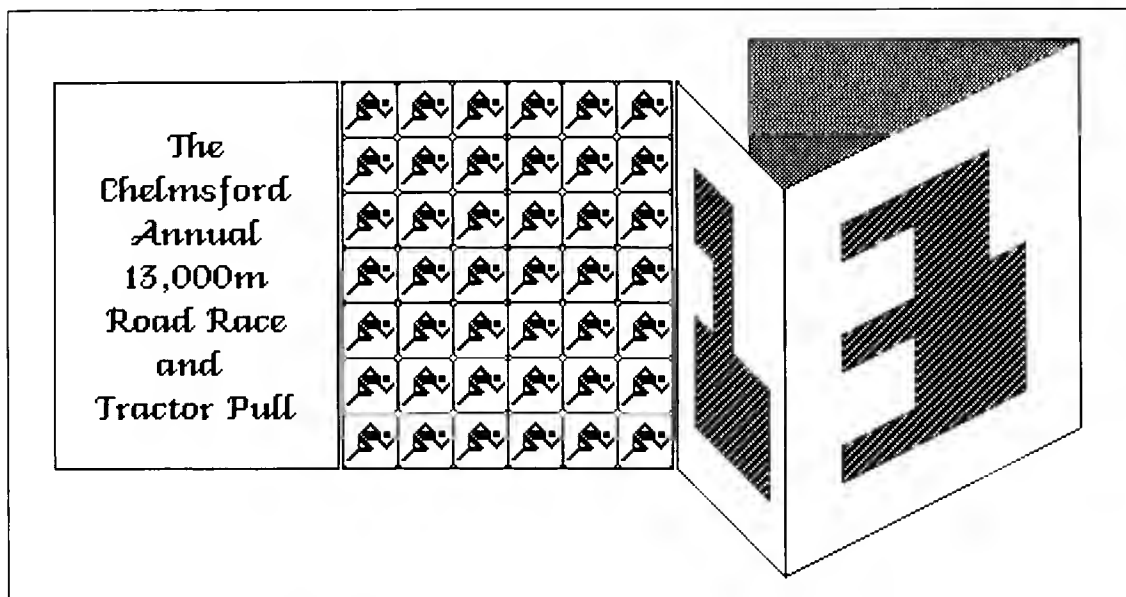


Illustration 4.8 Road race invitation

The three pages that lend a three-dimensional aspect to the graphic design by way of their folded look were produced by using Grid to hold the diagonal and vertical edges parallel.

The digits 1 and 3 were free-hand designs using Grid and the Line tool to outline the designs. Then the Paint bucket poured solid black throughout the finished pattern.

The fifth page, of which we see only a part of the back in shadow, was done by forming a vertical and horizontal line free-hand, although Grid or constraint would have helped.

We are increasingly impressed with the Macintosh, not only because it is so rich in features, but also because it is so rewarding and forgiving in its use. Its ability to allow you to *Undo* your mistakes, to *Erase* the too-long lines, to control every pixel with Fatbits, to duplicate, copy, and move entire images—these facilities are what artists dream of, and what amateurs like us must have so that we can approach free-hand art without reservations.

You have available here a tool to design icons of all kinds. Here's a small list that we generated with little thought:

1. Traffic signs
2. Types of books (Sci-fi, fiction, biographical, ...)
3. Building types

4. Vehicle types
5. Kinds of pets
6. Hobbies
7. Occupations
8. Musical categories
9. Kinds of vegetation (bushes, trees, vegetables, flowers, ...)
10. Foods
11. Tools
12. Toys
13. Types of airplanes, boats, cars, ...
14. Categories of television shows
15. Categories of movies

All of these, and many many more subjects, lend themselves to symbolic representation by icons. The use of icons as quick recognition symbols can only increase in our age of awareness of graphics, and the production of those icons is made ever more simple with the Macintosh.

A MACPAINT RECREATION

Tangrams have provided countless people with a simple recreation that mixes imagination with topological rigor. What are tangrams? What in the world is topological rigor?

Tangrams are images made from seven specially shaped tiles. The tiles, or *tans*, are really the shards of a larger square tile that has been broken into its seven components. Illustration 5.1 shows the unbroken square tan while Illustration 5.2 shows the exploded tan tiles and their 45° rotations.

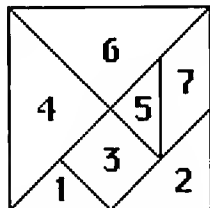


Illustration 5.1 Unbroken square tan

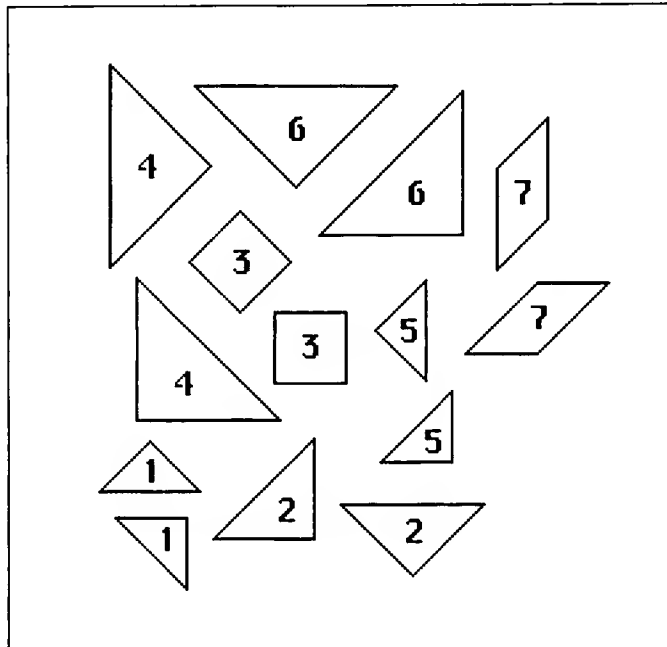


Illustration 5.2 Exploded tan tiles and their 45° rotations

Legend has it that ages ago a Chinese gentleman had a square tile. For some reason he dropped it, and it shattered into seven pieces. The Chinese proceeded to lay out the shards of tile into different patterns, with all tiles touching at some edge. He is supposed to have amused himself countless hours in this pastime activity, whereupon other people adopted the seven tans as a recreation. Whether this tale is in any way grounded in truth is highly debatable. We refer you to our book, *More Color Computer Applications*, John Wiley & Sons, 1984, pages 109-128 for a detailed discussion of the apocryphal origins of this activity.

Now, let's get to this business of *topological rigor*. The phrase implies that there are rules for laying out the seven broken shards, or tans, of the original square tile. The rules of the placement of those seven tans are: (1) all pieces touch at least one other piece along an edge; and (2) no piece may rest on or cover any part of another. These two simple rules provide a framework, or rigor, for making the final image; they are based on the *topology*, or the geometric layout while they are being positioned.

Consider the parallelogram, tile number 7 in the unexploded view in Illustration 5.1. You may wish to use it in a tangram but notice that its orientation is wrong. If the original tile had been black on top and white below, this piece would be white while the other six would be black. In other words, we have flipped over this tan. This is allowed in traditional tan manipulation. It is interesting to note that the parallelogram is the only tan whose orientation would not be reversed in this fashion if the turn-over process were against the rules. All other tans are symmetrical.

Application: Tangrams with MacPaint

The following procedures are based on tans that were derived from a square tile, but whose pieces were duplicated to provide 45-degree rotations so that you can manipulate them more easily with MacPaint.

The rules for producing tangrams are simple:

1. Use all seven tans of the original square tile.
2. Use no single tan more than once. If you use tile # 4 (large triangle with right angle pointing right) you cannot use the other # 4 (large triangle with right angle pointing left and down).
3. Do not place one tan over another. Have them meet at the edges only.
4. Do not isolate one or more tans. All must have at least one edge, or part of an edge, touching another tan.
5. With our version of the game, you may use any of the tan orientations available in the exploded view above, and their "flipped" images as provided using the Edit menu's Flip Horizontal or Flip Vertical or both. For example, tan # 7 can be used in any one of these orientations as shown in Illustration 5.3.

Consider the two examples in Illustration 5.4. The one at the left, the stork, is a legal tangram, which follows all of the rules above. The one at the right, which is also a stork, is not legal.

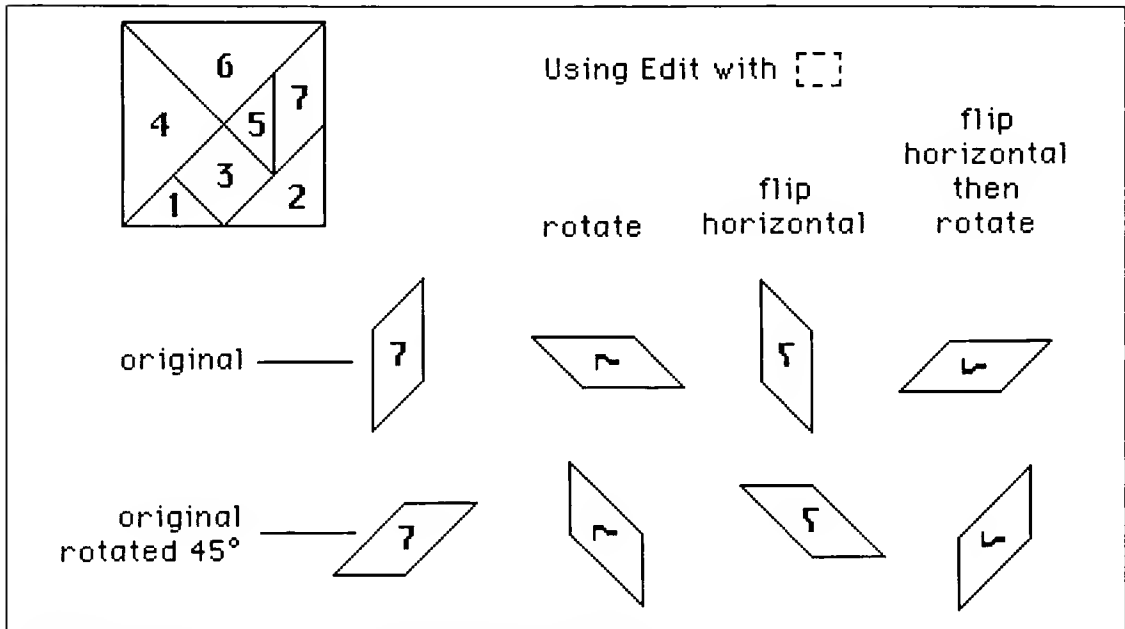


Illustration 5.3 Tan #7 shown in all possible orientations

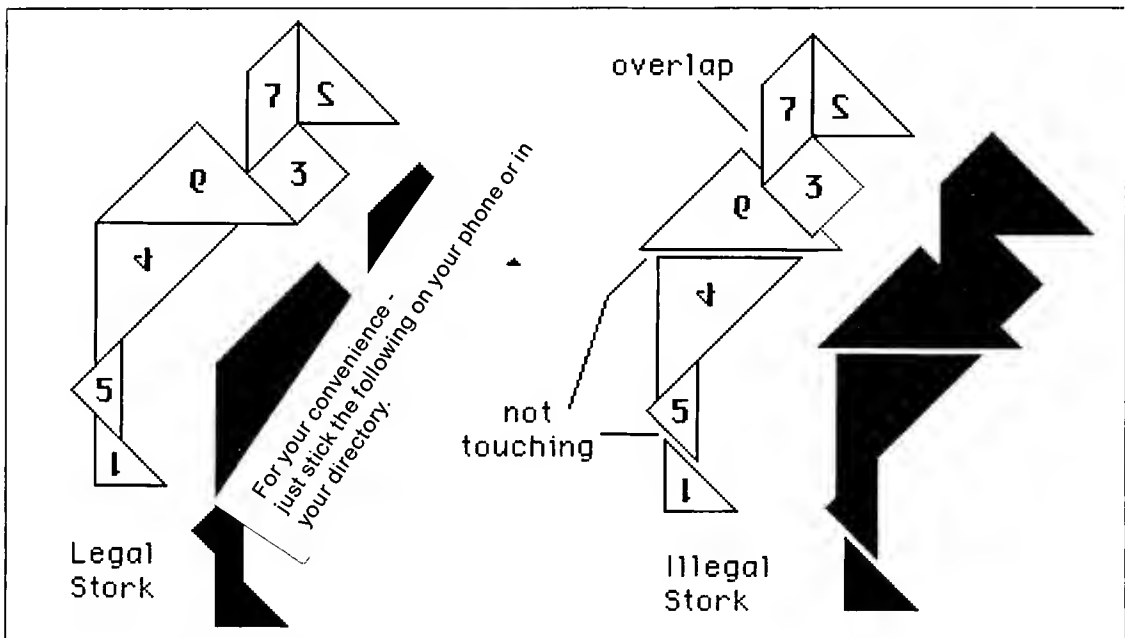


Illustration 5.4 Legal (left) and illegal (right) storks

Notice that when you manipulate tans with MacPaint using the flipping and shifting of all images we have provided, you cannot produce any orientations other than those in the original exploded tan field and their various vertical and horizontal flips. This keeps you from producing some tangrams that *would* be considered legal in the traditional pastime of tan manipulation, such as those produced with 30-degree rotations.

Using MacPaint to produce tangrams

To produce a tangram with our process of tan manipulation using MacPaint, you start with the field of 14 tans, (the seven original ones from the broken square, and their 45-degree rotations).

If you are starting from scratch, you must form the seven tans on your own. Select the rectangle, and using constraint and Grid draw a square. Then, using constraint and Grid throughout, divide the square as indicated in Illustration 5.5.

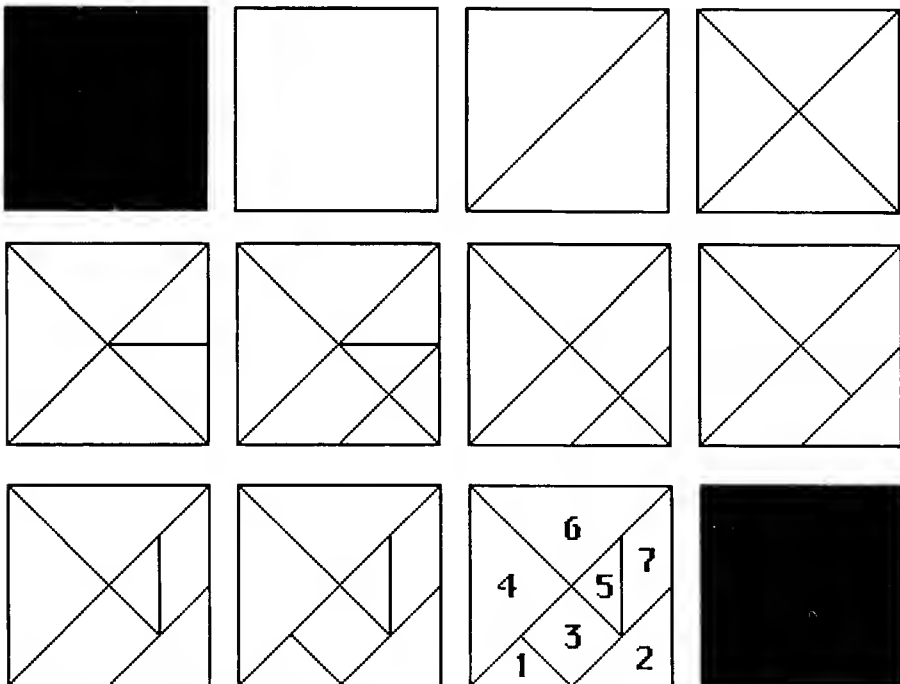


Illustration 5.5 Tan square construction

You can form the exploded image by either forming the tans separate from the square above, making certain that all formed tans fit exactly onto the original square tile, or by isolating each tan with the Lasso. If you use the latter method, you must leave behind the original square, so Lasso with the Option key pressed to make a duplicate image.

Once you have the exploded tans at the left of the MacPaint sketchpad area, you use the Lasso to drag a tan to the working area, the Marquee to isolate it for flipping and rotating, and Lasso again to position the tan exactly as you wish. Consider the procedure for producing the stork:

1. Lasso tan # 2 (the # 2 tan that has the original orientation from the square tile, not the # 2 with the apex down) and drag it to the workspace at the right of the sketchpad.
2. Select the marquee and enclose this tan. Flip horizontal and position it at the top right of the workspace.
3. Lasso tan # 7 (original orientation) and drag it over to the previously positioned tan # 2, carefully placing it so that the two edges fit precisely over each other. If you let go of the mouse and it doesn't look right, use *Undo* from the Edit menu.
4. Lasso tan # 3 (original) and drag it to its final position.
5. Lasso tan # 6 (original) and drag it into the open space.
6. Select the Marquee and enclose this tan. Flip vertical.
7. Lasso this tan and drag it into its final position.
8. Lasso tan # 4 (the 45-degree rotated one, the one with only one diagonal, not the original which has two diagonals) and drag it into the open space.
9. Select Marquee, enclose tan, flip vertical. Lasso and drag to its final position.
10. Lasso tan # 5 (original) and drag it to its final position.
11. Lasso tan # 1 (45-degree orientation, with one diagonal), flip both horizontal and vertical, and drag it to its final position. If you wish, you can rotate it twice and achieve the same result.
12. Lasso the entire finished stork, duplicate it (option-drag) in the available open space.
13. Select the black pattern from the palette.
14. Using the paint bucket, fill in all seven tans to produce the final image of the stork.

If you were to paint the stork some pattern other than black, you would have to remember that the various numbers that were in the tans would show through. There is a feature of MacPaint that *washes* paint throughout an area, so that the

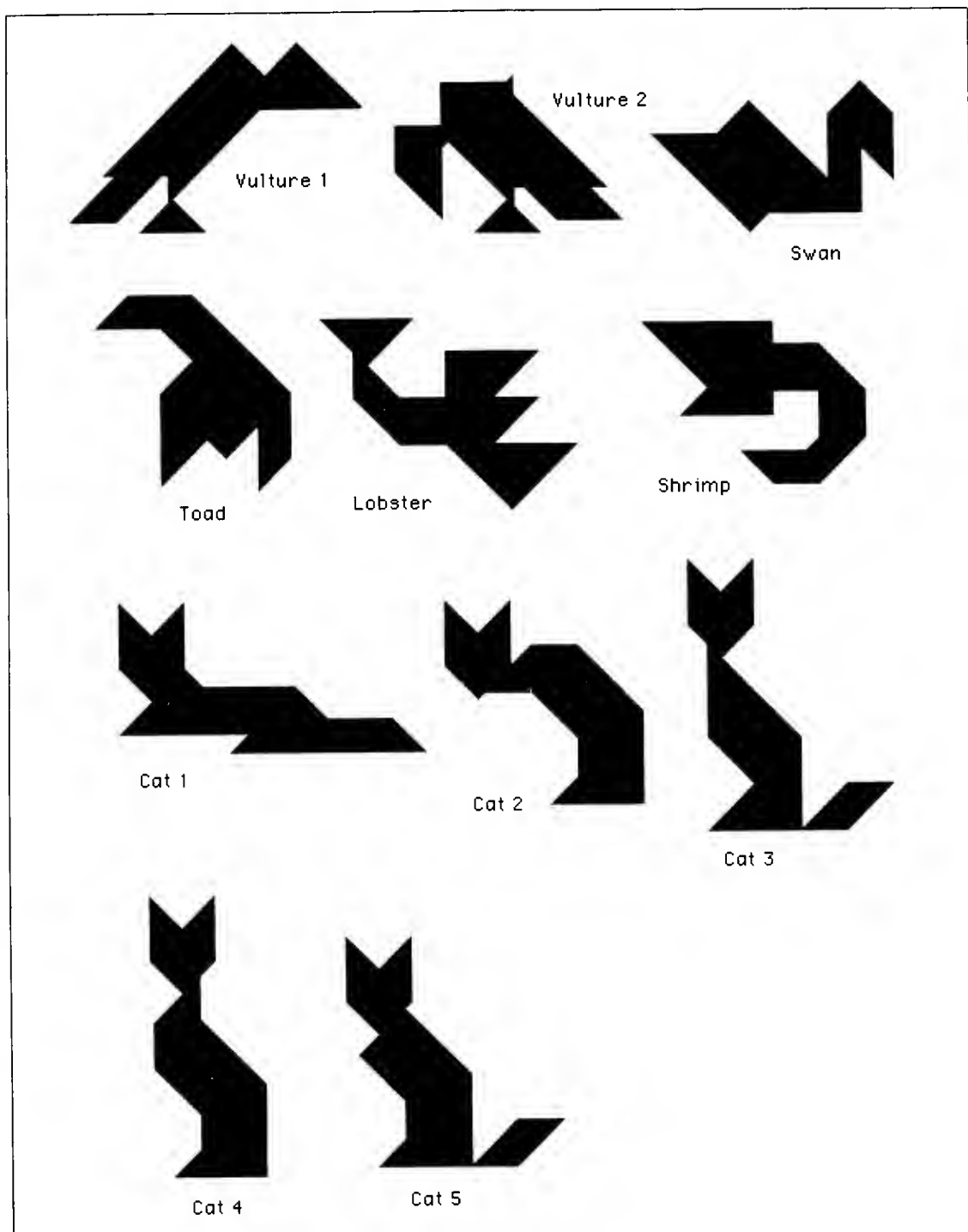


Illustration 5.6 Various Tangrams

paint covers everything within the enclosed area. To do this, use the paint bucket but hold down the Shift key while pouring the paint (clicking). This is suggested as a possibility, although we think you will find the pure black tangrams most effective in representing the image they were intended to depict.

As a last remark on tans and tangrams, we include in Illustration 5.6 a few of the many tangrams that you can produce on your own. We discovered these images and hundreds of others in several books that might interest you.

The list of references we include below should get you started on the subject of tangrams if you wish to pursue this topic.

Elffers, Joost, *Tangram: The Ancient Chinese Shapes Game*, Translated by R. J. Hollingdale (Penguin Books, New York, 1978).

Grillo, John, and J.D. Robertson, *More Color Computer Applications*, pages 109-128 (John Wiley & Sons, New York, 1984).

Johnston, Susan, *Tangrams ABC Kit: 122 Puzzles With Two Complete Sets of Tangram Pieces* (Dover Publications, Inc., New York, 1979).

Read, Ronald C., *Tangrams: 330 Puzzles* (Dover Publications, Inc., 1965).

Van Note, Peter, *Tangrams: Picture-Making Puzzle Game* (Charles E. Tuttle Company, Vermont, 1966).

We're not through with tangrams yet. Chapter 11 rediscovers the topic, only this time using the power of Microsoft BASIC and the Macintosh mouse to manipulate the tans. We think you will find the combination of these two chapters to be an interesting contrast between a canned application like MacPaint and a more flexible program that simulates some of MacPaint's fine features.

PROGRAM PLANNING

During the past decade or so, a quiet revolution has taken place in programming shops everywhere. Programmers have recognized the value in program planning, and they have adopted some approaches that were unheard of fifteen years ago. We are referring to *top-down design* and *modular programming*.

Consider for a moment the traditional order of operations in developing a program:

1. Understand the problem.
2. Develop the algorithm for solving the problem.
3. Draw a program flowchart.
4. Code
5. Debug and test
6. Document

There is no argument with the first step. After all, unless you understand what it is that the computer must do for you, there is no way that you can successfully write the program to solve the problem. The real difficulty comes with the second step. Here you are supposed to describe in some unambiguous, finite, effective way exactly how the program should be designed. That's some job!

The third step, drawing the flowchart, is supposed to lay out the program's logic so that the fourth step, coding, isn't so hard. Unfortunately, that flowchart depends upon your complete understanding, in great detail, of the algorithm or algorithms developed in the previous phase, and all too often that isn't the case.

Notice that the fifth step includes debugging, which assumes errors on your part. If there is any value to the new ideas in top-down, structured, modular programming and program design, it is the fact that your programs will contain far fewer errors, and the errors that they do contain will be much easier to detect and remove.

So what is this plan? How do you develop your program using top-down design principles? What are the differences in the order of operations, and in the operations themselves?

First, a Grillo-Robertson rule of thumb: Do *something* on paper first, before even turning on the machine. The most destructive thing you can do to a program, and the most wasteful use of your time, is to start coding without a plan. Even if your plan on paper consists of a set of several simple steps written in English, at least you have in front of you an ordered plan of action, with a beginning, middle, and end.

Top-down design

The phrase top-down design implies a stepwise approach to the problem's design: First you consider it from a distance, or from its most abstract form, then at the deeper levels involving more and more detail. For example, the goal of a program might be to display a rectangle on the screen.

No method is described in this statement, only the ultimate goal of the program. A second step in the top-down design might be a simple list of activities, in English, to accomplish the goal.

1. Get length and width from user.
2. Check to see if it fits the screen.
3. Draw the rectangle.

Notice that this second, more detailed step, includes some information about the method to be employed in solving the problem, and it also includes the order of its most important steps. Even further detail is present in the third step of the same program.

1. Clear the screen.
2. Get length and width L and W.
3. If $W + 20$ greater than screen width or $L + 20$ greater than screen length goto step 2.
4. Clear screen.
5. Draw line down from (10,10) to (10, $W + 10$).
6. Draw line down from ($L + 10$,10) to ($L + 10$, $W + 10$).
7. Draw line across from (10,10) to ($L + 10$,10).
8. Draw line across from (10, $W + 10$) to ($L + 10$, $W + 10$).

Progressing from this plan to BASIC is trivial, and that's the idea behind top-down design—to turn large, seemingly insoluble problems into several small, easily designed, coded, and debugged *modules*. The idea is as old as Plato, and very effective.

Steps in Program Planning

The overall plan for designing any program should include the steps shown in Illustration 6.1.

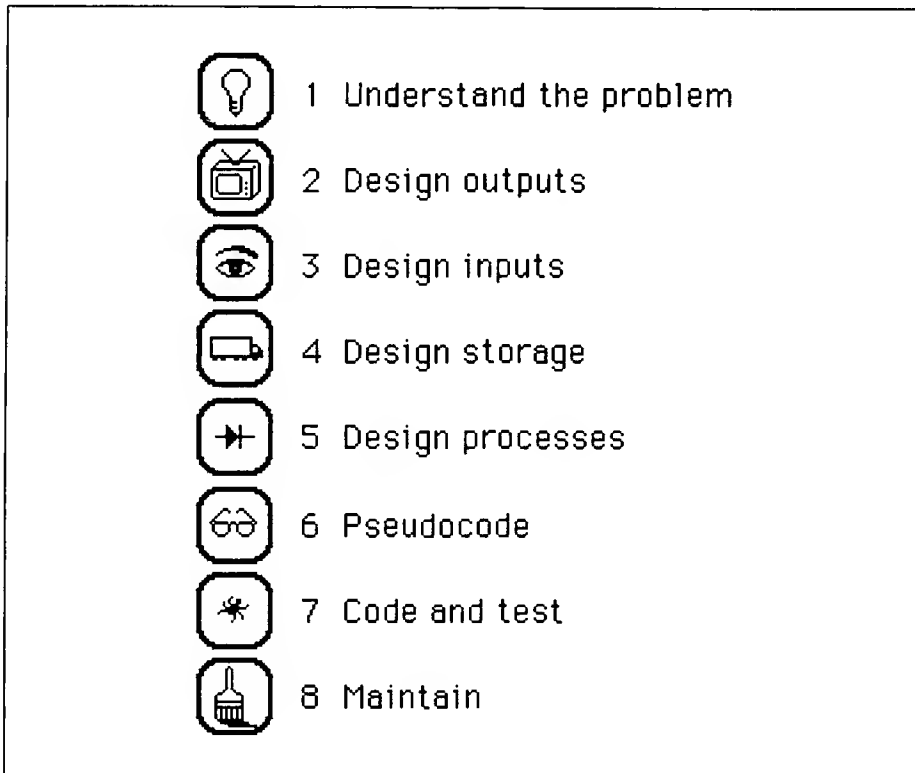



Illustration 6.1 Steps in program planning

Let's take each of these steps in turn to point out the advantages of this technique.

-  **Understand the problem:** There's no way to avoid this step in any scheme for writing programs. What does the user really want from the computer? You have to understand what is involved in the overall interaction of computer and user. One of the best ways to understand the problem better is to try to take the place of the user of your program. Depending on the user's level of sophistication, you may be able to get by with minimal dialog, or you may have to do a lot of hand-holding and guide the user in all operations. The common term for this approach is *user friendliness*.



2. **Design the outputs:** Again, depend on the user of the program to provide you with the proper answers here. Often, you will be the end-user of your own program, so the process is made a little simpler. In either case, you *must* design all of the program's outputs, both screen and printer. Some people try to bypass this step, or to minimize it. It is by far the most crucial step of all in graphics programming, and should be executed with care and detail. You don't have to go to the detail of calculating answers or even drawing the pictures, but you should anticipate the size (how many digits accuracy?), the type (real or integer?) and the format (commas? dollar signs? etc...) of numeric output; you should plan a rough layout of your graphic output; and you should be able to plan verbal requests and responses.



3. **Design the inputs:** In years past, when batch systems using punched cards were popular, this step would have included the formatting of the cards to be entered as data into the system. Now, because most of the computing is done on an interactive basis with both the computer and the user entering into a dialog, it is important to plan that dialog carefully.

When you design a screen in the step above, you should consider whether that screen also displays some user inputs, as is often the case. For example, a data entry module may have a full-screen display with blanks in reverse video indicating where the user will enter data.


It is important to anticipate user responses, especially whether they are correct or not. If the user goofs and enters a value for a coordinate that is out of bounds on the screen, how do you plan to deal with it? Later on, you'll have to code data entry modules to include some bypasses for all user errors. Now is the time to start thinking about this problem.



4. **Design the storage:** In Steps 2 and 3 above, you have been thinking about responses to requests, whether they are from the computer or from the user. It may be, at its most rudimentary level, a single digit response by the user to a request from the computer to select an activity from a menu. In any case, it is either an *entry* by the user into a memory location for the computer to process, or it is the *display* of a computed or stored result.

The entry is *placed into* a memory location, and the display is *copied out of* a memory location. That memory location must have a name in the program, and now is an ideal time to begin to select your variable names.

In programs that deal with files, you decide the type of file the program must use—whether it is sequential, direct access, ISAM, or some combination. And you need to describe the layout of the records in the file. We recommend that you get a copy of our book in this series, *Data and File Structures on the Macintosh* if you are serious about file storage.

5.  **Design the processes:** As is the case in all of the steps above, you still haven't touched the computer. This phase of the design of your program begins to deal with the piecewise outline of all modules, and their interrelationships.

The most useful tool in this step is the *hierarchy chart*, which in many ways resembles an organization chart for a business. Such a chart has several features that describe the top-down design of your program, or system of programs. First, it is made up of rectangles that signify program modules. Second, every program module must be accessible only from a module above the one in question. Third, the lower module on the hierarchy chart is subordinate to the one above in its function.

You can develop this chart showing all modules of your program and their relationships in two or more steps, if the program is complex. For example, suppose you are writing a system for producing graphics in an architectural environment. The first hierarchy chart could show only two levels of detail, as sketched in Illustration 6.2.

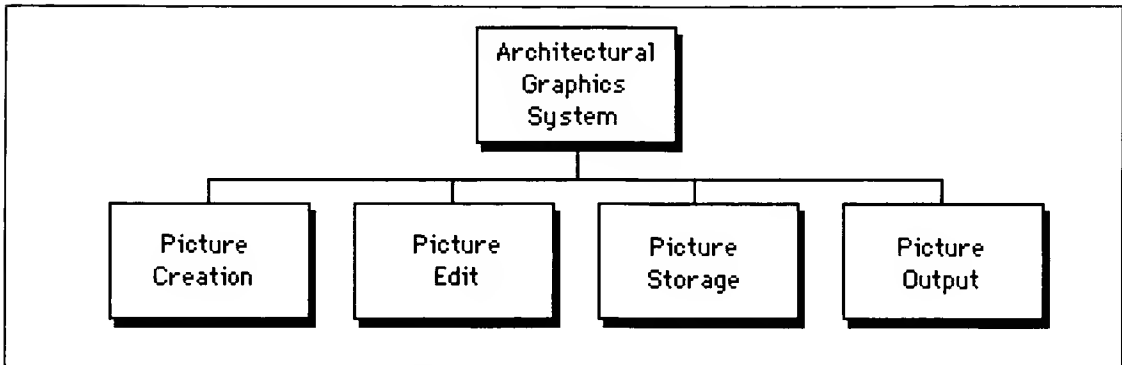


Illustration 6.2 Hierarchy chart, architectural system

The second stage in the development of the hierarchy chart could show a third level of detail, thus indicating the major programming modules. (Illustration 6.3)

Developing this chart even further, you could take one of the third-level modules and describe it in great detail. Now, you are at the stage of understanding the interrelationships of all modules, large and small, and you can begin to visualize the subroutines.

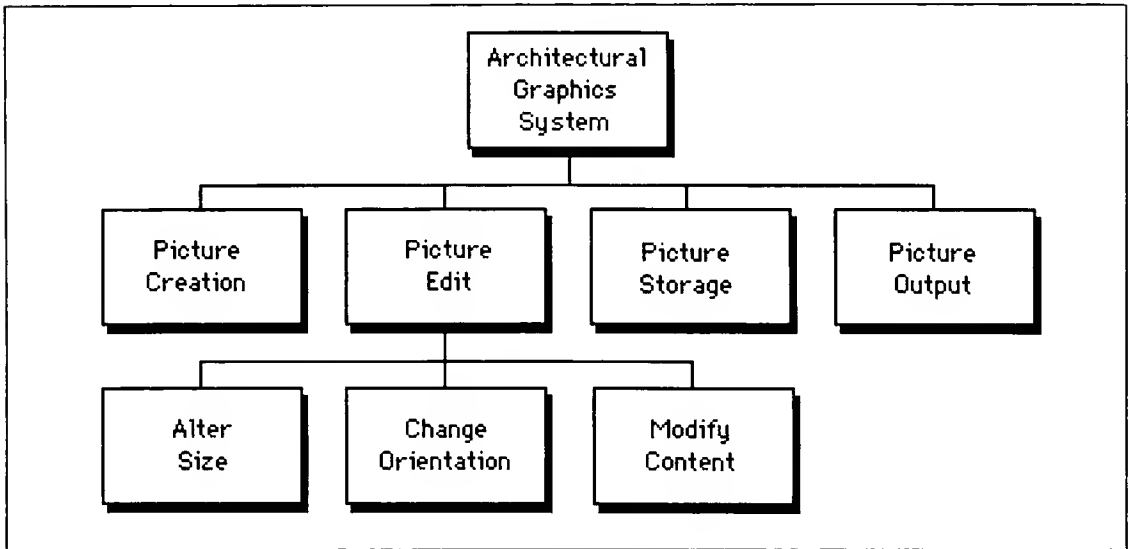


Illustration 6.3 Portion of three-level chart



6. **Pseudocode:** With your detailed preparation in all of the steps above, you now have a firm grasp of the program's variable names, the file layouts, the computer's printed results, the user's responses to requests, and even the interrelationships between the main and subordinate modules of the program. Now you can begin to outline each module in steps small enough to be translated easily into single statements or small segments of code. *Pseudocode* is a step-by-step English language description of the problem to be solved.

This process of outlining the programs before writing them is critical. It takes the place of drawing a flowchart in the old-fashioned program planning process. In most ways it is far simpler, and more effective as a means of describing code to be written. A good understanding of *structured programming* helps a great deal in this step. We describe the essentials of program structures in the next section of this chapter.

The easiest, and perhaps the most effective, pseudocode is English as you use it. You don't have to remember any funny words. All you do is outline your program segment in small steps, designing loops and tests as you go. For example, a small program module to draw a sector in a pie chart could be pseudocoded this way:

- a. Convert percentage of whole graph into an angle; call it SECTOR-SIZE.

- b. Draw line from XCENTER,YCENTER to STARTANGLE + SECTORSIZE.
- c. Set INDEX to STARTANGLE.
- d. While INDEX is less than STARTANGLE + SECTORSIZE do:
Add .05 to INDEX Plot point at (XCENTER + RADIUS * COS(INDEX),
(YCENTER - RADIUS * SIN(INDEX))
- e. Select sector pattern.
- f. Fill sector with pattern.
- g. Set STARTANGLE to STARTANGLE + SECTORSIZE.
- h. Return.

Not all pseudocode contains variable names, nor does it all have as much math as the example above. Sometimes it is sufficient to outline the steps in even more English-like steps, such as the pseudocode below that describes the algorithm for a simple exchange sort.

- a. Start a loop with the first element of the list, going to the second from last in the list.
- b. Start a second loop, with the index going from one more than the first loop's index to the last element.
- c. Compare the two elements that the loop indices point to.
- d. If the first is more than the last, swap them.
- e. End of second loop
- f. End of first loop.
- g. Return.



The BASIC code that results from this pseudocode is:

```

1000 FOR I = 1 TO N - 1
1010  FOR J = I + 1 TO N
1020    IF X(I) > X(J) THEN SWAP X(I),X(J)
1030  NEXT J
1040 NEXT I
1050 RETURN

```

If there is any rule to guide you in the process of pseudocoding an algorithm, it is this: Keep it sequential and simple.

7.  **Code and test:** Finally, you can begin to write statements in BASIC, and to run the modules you have coded. Start at the very top of your program, and code the main module first. Keep it short and flexible, because you may have to add or modify its code later. Be sure to provide GOSUBs to the modules you want to code and test next. If you code each module or subroutine to be visible in its entirety on one screen, you're on the right track.
8.  **Maintain:** Any program worth writing (unless it is a one-shot test) is worth maintaining. This may involve nothing more than some occasional user-proofing so that the program can deal with a wider range of user responses. It may be an extensive rewrite or the addition of several large modules, a substantial enhancement to the old program.



Structured Programming

All programs are made up of primitive *structures*, or small elements, somewhat corresponding to the nouns, verbs, and adjectives of a spoken or written language. In the case of a programming language—any programming language—those elements or structures are:

1. Sequential structures—code that follows one step after another. In a programming flowchart (Illustration 6.4), a sequential structure is shown as one or more processes in a line:
2. Decision structures—two-way branches.
The flowchart symbol for a decision structure is the IF test (Illustration 6.5).

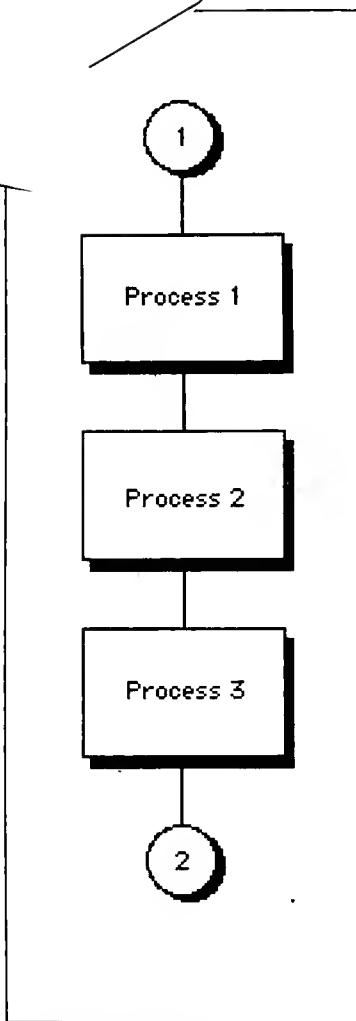


Illustration 6.4 Sequential structure

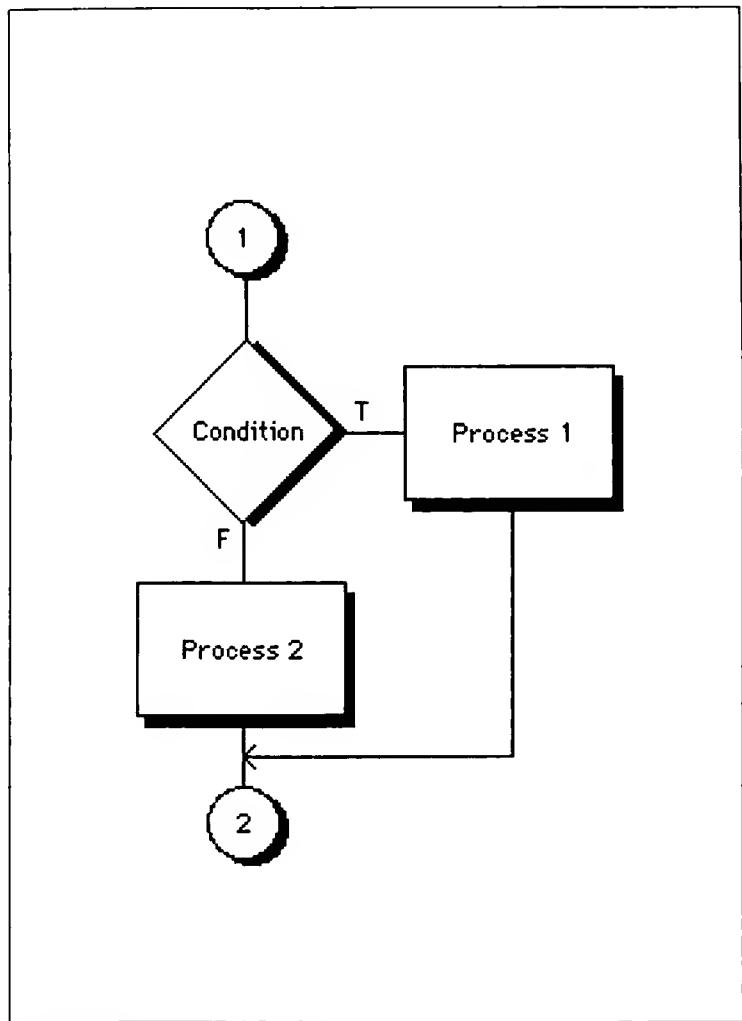


Illustration 6.5 Decision structure

3. Loop structures—either a DOWHILE or a DOUNTIL
The DOWHILE structure is shown in the program flowchart in Illustration 6.6.

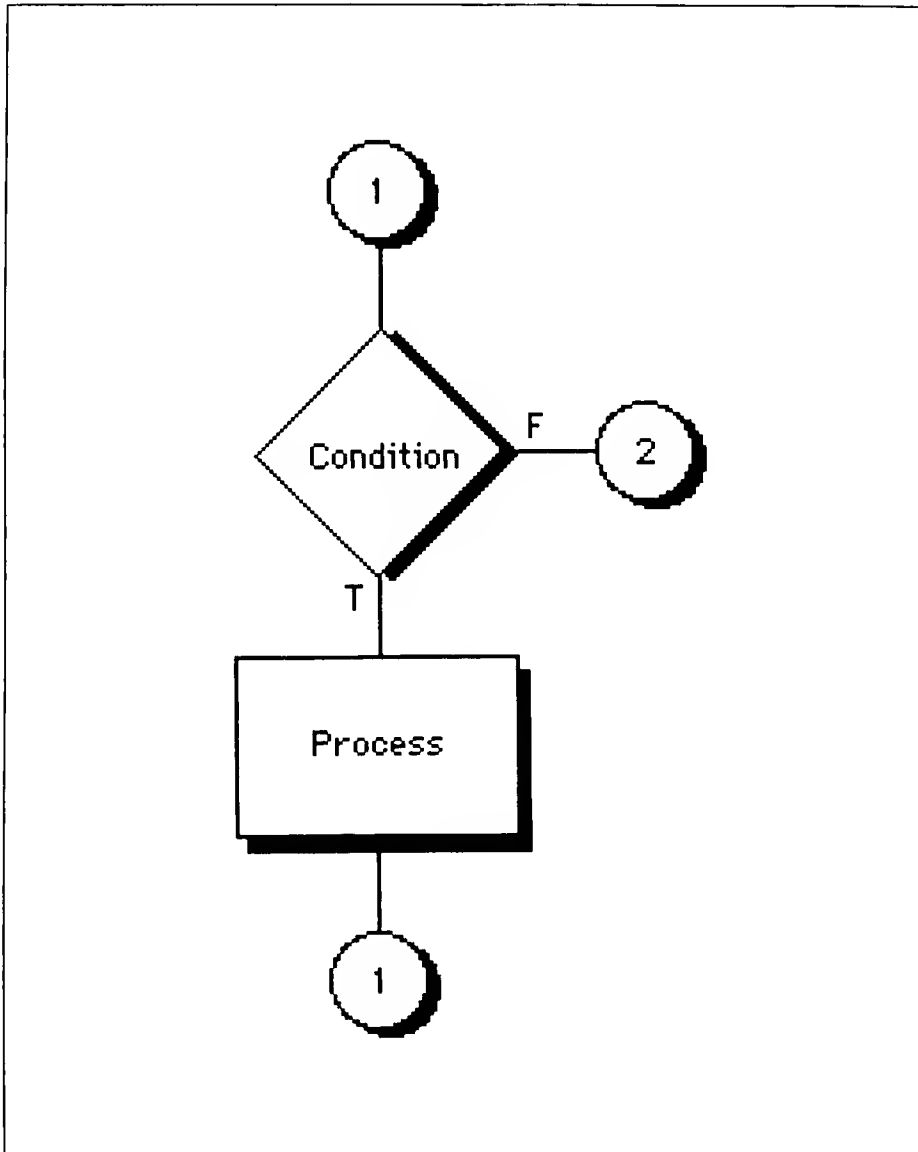


Illustration 6.6 DOWHILE structure

The DOUNTIL structure is shown in Illustration 6.7. Note that this structure forces one iteration of the loop regardless of the initial condition of the element that is tested.

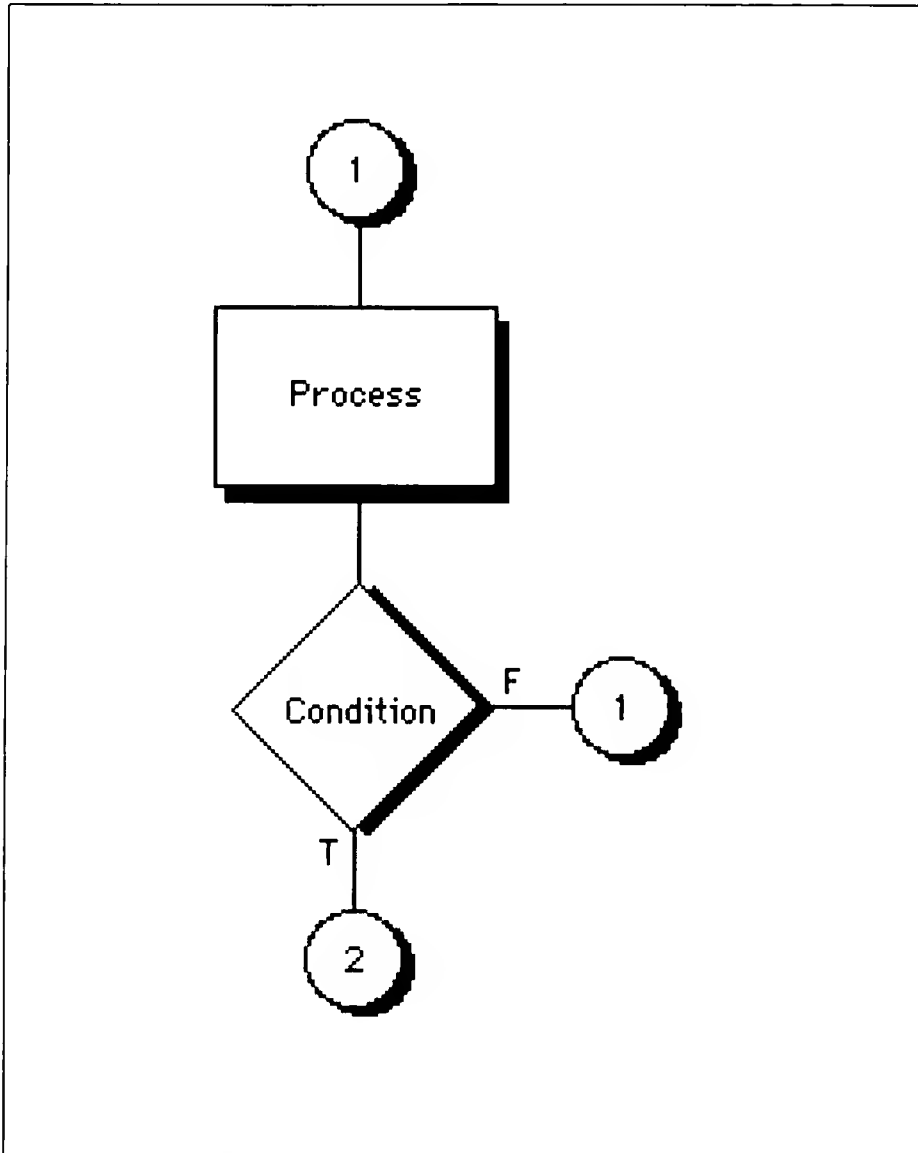


Illustration 6.7 DOUNTIL structure

4. Case structures—multi-way branches, as provided by the BASIC ON-GOSUB or ON-GOTO statement (Illustration 6.8).

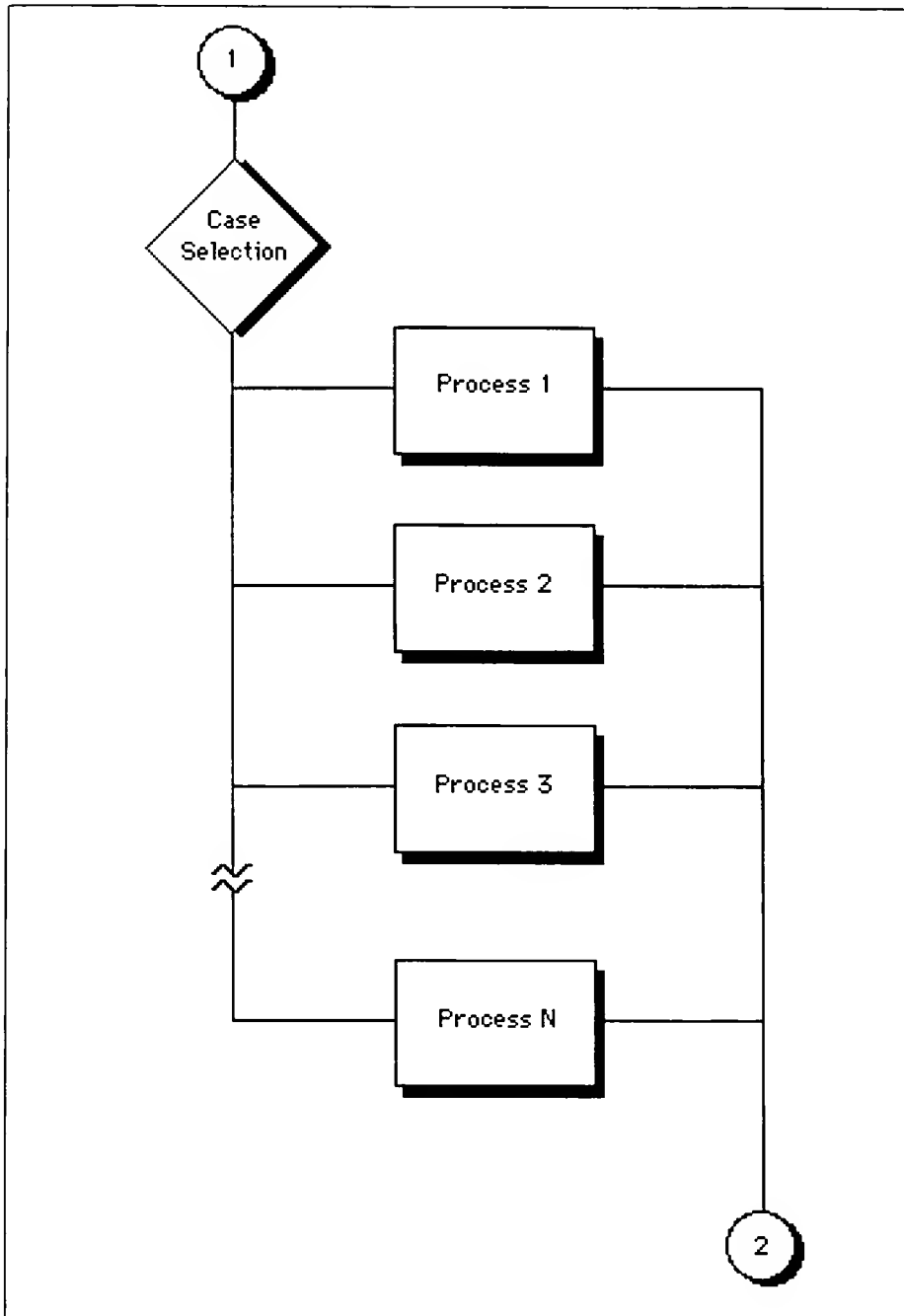


Illustration 6.8 CASE structure

These four simple elements of any program can be used to understand the logic of a program at its most primitive level. They reduce the programs to the level of one or several statements tied together into a unified, small increment of logic.



Menus and Submenus, Main Programs and Subprograms

To the user, a program is often best represented as a series of menus, each with its list of activities. When the user starts the program, she sees a list of the major activities that the program or system of programs can manage. The selection of one of these main activities often provides the display of a second, subsidiary menu, or submenu. This form of organization of a program is user-friendly. It deals with the subject in the user's terms, in a non-threatening way. Consider Illustration 6.9, for example, which shows a series of menus and submenus for an architectural sketching system of programs:

MAIN MENU—HOUSE PLAN SYSTEM

1. Build new plan
2. Add to existing plan
3. Modify a plan
4. Print house plan
5. Estimate costs
6. Utilities
7. Stop

Illustration 6.9 Main Menu

SUBMENU 1—Build New Plan

1. List names of existing plans on file
2. Name this plan
3. Describe this plan — one or more floors, scale, ...
4. Return to Main Menu

Illustration 6.10 Submenu 1 — input

SUBMENU 2—Add to Existing Plan

1. Retrieve existing plan's description
2. Add detail
3. Return to Main Menu

Illustration 6.11 Submenu 2 — more input

SUBMENU 3—Modify a Plan

1. Select plan
2. List plan details
3. Select a specification
4. Modify
 - a. Change a plan's contents
 - b. Change plan specifications
5. Delete an area
6. Return to Main Menu

Illustration 6.12 Submenu 3 — processing module

SUBMENU 4—Print Plan

1. Print a specific area
2. Scale and print entire plan
6. Return to Main Menu

Illustration 6.13 Submenu 4 — output module

SUBMENU 5—Estimate Costs

1. List presently filed costs
2. Change listed costs
3. Calculate
4. Rank estimated costs
5. Return to Main Menu

Illustration 6.14 Submenu 5 — processing module

SUBMENU 6—Utilities

1. Display memory variables, contents
2. Sort files
3. Change scale
4. Dump all fields of a plan
5. Return to Main Menu

Illustration 6.15 Submenu 6 — processing (utilities)

The program that you write can be structured in such a fashion as shown in the sequence of menus in Illustrations 6.9 to 6.15. In fact, it is good practice to structure your programs this way to keep them modular, so that small segments can be developed and tested one at a time. We have discussed this practice before, so let's see what a program could look like if it is subdivided into main and sub-modules.

The central idea in developing program modules is the *subroutine*. In most programming languages, the subroutine is compiled separately from the main program, which further isolates it. In BASIC, however, both the subroutine and the main program are together, and they use common variables. This somewhat unconventional approach to subroutines has advantages and hazards. Certainly it is handy to be able to see both sections of source code at once. But it can become burdensome to remember all of the variables that the subroutine alters in its execution, so as not to use those variables in the main program.

We have developed a skeleton program that can serve as a shell for most large programs. This shell design depends on the program dealing with user menus, so the two concepts go together nicely. The overall, most generalized, structure is shown below:

General Structure

1. Main Program
 - a. Initialize variables
 - b. GOSUB Main Menu display routine
 - c. If (user response) then goto 1.b.
 - d. If (user response) then stop
 - e. ON (user response) GOSUB Subroutine-1,-2,-3,...
 - f. GOTO 1.b.
2. Tool Subroutines -- callable from anywhere
 - a. Generalized list
 - b. Error handling
 - c. Menu display
 - d. ...

3. Subroutine-1

- a. Display sub-menu1
- b. If (user response) is <RETURN> then return
- c. ON (user response) GOSUB Subroutine-3a,-3b,-3c,...
- d. GOTO 3.a.

4. Subroutine-2

- a. Display sub-menu2
- b. If (user response) is <RETURN> then return
- c. ON (user response) GOSUB Subroutine-4a,4b,4c,...
- d. GOTO 4.a.

5.

This highly flexible skeleton for any complex program can be modified or extended in different ways. For example, instead of going to a main subroutine, the program can branch to a CHAIN statement that will transfer control to an entirely different program. In this way, the internal memory limitations of the Macintosh are no longer bothersome.

In the next chapter, we will explore some of the excellent features of Microsoft BASIC on the Macintosh that allow very large systems to be developed in a modular fashion, such as CHAIN, MERGE, COMMON, and mouse-driven menus.

MAC, THE USER, AND BASIC



Introduction: In Defense of BASIC

BASIC has received considerable criticism, at least in part because of its historical roots in educational settings as an elementary language that is simple to learn. In fact, during the last ten years it has matured into a successful small-business and personal computer system language. What are these criticisms, and how have most of them been overcome?



1. **Non-standard subroutines:** Most high-level languages have the built-in capability to separately compile subroutines that calls the main program (or another subroutine). This has two advantages: first, the subroutine can be developed, tested, and compiled during one phase of the system's development; second, the subroutine's variable names are totally independent of the calling program. They are *local* variables, whose names can be used in other subroutines, or in the main program. BASIC uses *global* variable names in its subroutines, because the subroutine is no more than a section of code in the program from which control can return to the statement below the GOSUB. This means that you must take care when writing a subroutine to invent variable names that are not used in the other programs that might call the subroutine.

The American National Standards Institute (ANSI) has recently proposed several changes to BASIC, one of them being the concept of true subroutines, that can pass variables and return variables, and that can use local variables with the same names as those found in the calling program with no effect. Microsoft BASIC has a CALL statement, but it is reserved for calling subroutines in machine language only. We use a programming standard of our own, which may be more accurately described as a stylistic habit rather

than a standard. This style names locally used variables in subroutines with a certain suffix, say the digit 7 or 8. Thus a subroutine's variables could be W8, T8, and B8, while its calling programs would never use any variable names ending with the digit 8. This style is effective but crude.



2. **Slow, interpreted code:** In many business systems and in most personal systems, this has little effect, because BASIC is still fast enough for most purposes. However, when a SORT is executed, or when you are developing a graphics program that simulates animation, there is a significant slowdown in execution speed. This slowdown is especially prevalent in hardware systems that are driven by 8-bit microprocessors, such as the older personal computers (APPLE-II, Radio Shack TRS-80, Commodore 64, or IBM PC, for example).

The Macintosh uses the 16-bit MC68000 microprocessor as a CPU, so it is somewhat faster. However, any 8-bit-based personal computer that runs a compiled BASIC program will execute its program between 20 and 100 times faster! More and more vendors are supplying BASIC compilers to overcome this difficulty, and perhaps some day the ANSI standard will demand compilers as a standard language processor. Until then, there are enough clever tricks that programmers can use to significantly speed up execution.



3. **Lack of advanced language features:** This criticism often comes from those programmers whose primary language is Pascal, or COBOL, and even from some who still write in FORTRAN. Of course, BASIC doesn't have some of the features of these languages. But neither do they have some of BASIC's bells and whistles. So much of this type of criticism is based on what a programmer is used to, that it becomes a sort of chauvinistic trademark.

Some newer languages based on Pascal, but containing improvements such as Ada and Modula-II, compare favorably with BASIC in terms of language features. In truth BASIC has almost all of the programming structures (IF-THEN-ELSE, ON-GOTO and ON-GOSUB, integer, real, double precision, and string variables, and sequential and direct access file manipulation, for example) that are the earmarks of a good language. On top of that, BASIC's excellent string manipulation functions are singularly good, and they are easy to use.

Microsoft's BASIC as developed for the Macintosh is so much like the BASIC it has prepared for the IBM PC, the HP-150, the TRS-80, and the many other micros, that it is in essence the *de facto* standard for small-business and personal computers. The Macintosh BASIC has, in addition to this excellent core of features, access to the Mac's toolbox of machine-language routines

in the Quickdraw section of its ROM. This feature alone gives BASIC an unusually flexible power. This book will develop many programs that use these routines.



Program Development Tools

Microsoft BASIC has several useful tricks up its sleeve when it comes to developing large systems of programs in which there are several separate programs developed as *system modules* and each of these modules is accessed by either a single *main program* or by all other system modules. Consider these two system designs:

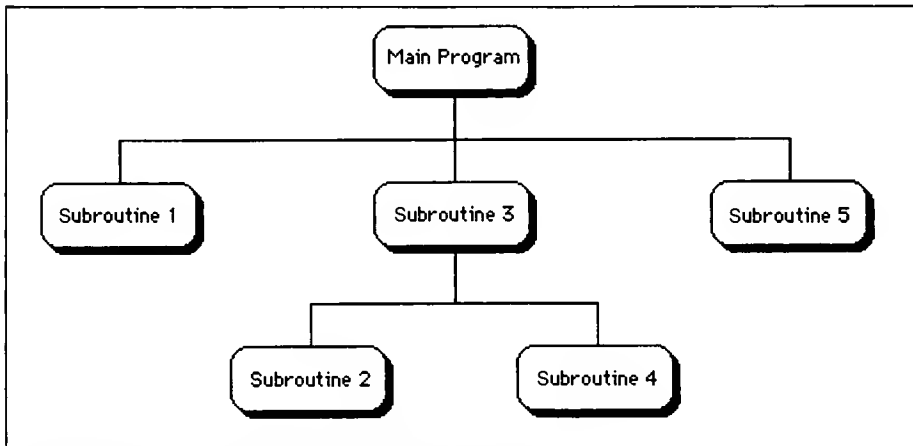


Illustration 7.1 Top-down system design chart

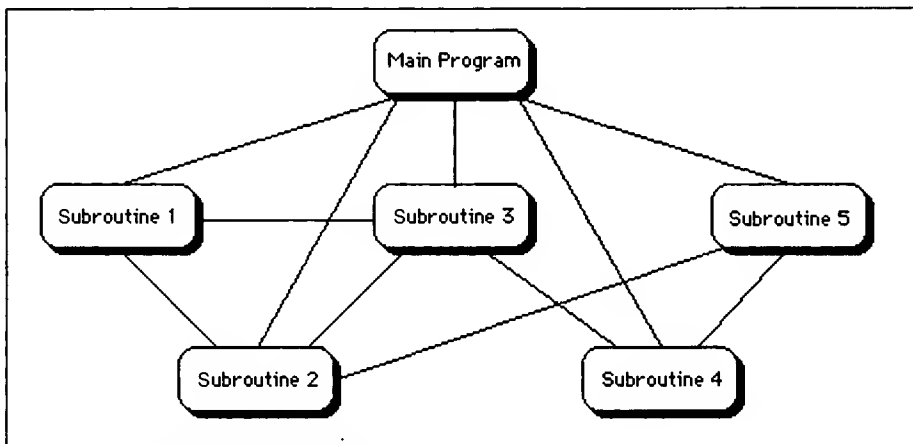


Illustration 7.2 Network design chart

In the top-down design, every time a subroutine is called, it must return to the main program module in order to access another subroutine. In the network design, any module can call any other, and the source of the passage of control is not important. Both system design structures are achievable in Microsoft BASIC, but we will deal mostly with the first, *top-down design*.



CHAIN

The CHAIN command in BASIC is used to pass control to another program. We are not talking about a segment of the same program here, but an entirely different program saved as a different file. We will summarize the CHAIN command. Refer to the Microsoft BASIC Interpreter Manual for a more detailed explanation.

Example 1: (Line 210 in the program called MAIN)

```
210 CHAIN "SUBPROGRAM ONE"
```

This command transfers control to the first line of the file called SUBPROGRAM ONE. In addition, it keeps open all files that were open in MAIN. Its effect is identical to

```
210 LOAD "SUBPROGRAM ONE", R
```

Example 2: Suppose some variables have been established in the program MAIN, and you wish to keep them active in the program called "SUBPROGRAM TWO".

```
210 CHAIN "SUBPROGRAM TWO",,ALL
```

These examples transfer control while keeping all variables active as well. Also, it keeps the files open during the transfer. Again, control resumes at the first line of "SUBPROGRAM TWO".

Example 3: If you wish to keep the variables active, the files open, and transfer control to line 700 of the file called "SUBPROGRAM THREE", you would issue this instruction in the program:

```
210 CHAIN "SUBPROGRAM THREE",700,ALL
```

If you know that the line you want to transfer to in the file "SUBPROGRAM THREE" is 100 times the value of P, you could write either of these:

```
210 CHAIN "SUBPROGRAM THREE",100*P,ALL
```

Example 4: To bring in a second file as an *overlay* to the first, that is, to wipe out all common lines in the first so as to reduce the size of the program in memory and yet keep some of the original lines in the final version, you can use the MERGE option. Suppose you want to keep all lines below # 100 in the MAIN program intact, but to replace all lines 1000 and above with those from the file SUBFOUR.

```
210 CHAIN MERGE "SUBFOUR",1000,ALL,DELETE 1000-9999
```

This example shows the typical instruction used to create an overlay. First, the lines 1000-9999 in MAIN are deleted, although if some of them were executed and assigned values to variables, they are kept active. Then, the file SUBFOUR, previously saved in ASCII with a command such as

```
SAVE "SUBFOUR",A
```

is brought into memory and merged with all of the remaining lines of MAIN. Execution of the program resumes at line 1000 of this newly created program.



COMMON

There are many times when you might want to CHAIN to another program, but keep only some of the variables active in memory. This means that you can't use the ALL option in the CHAIN instruction. Instead, you must use the COMMON statement within the MAIN, or calling, program. For example, if you want to keep the simple numeric variables A, B, and C; the string array X\$; and the integer array T% active in the program SUBFIVE, and you want to have execution begin at line 4270 in SUBFIVE, this is what you do:

In the program MAIN, preferably close to the beginning, you place the COMMON statement:

```
10 'FILENAME: "MAIN"
.
.
.
50 DIMENSION T%(200), X$(30), .....
.
.
.
80 COMMON A, B, C, X$( ), T%( )
.
.
.
210 CHAIN "SUBFIVE",1000
```


One note of caution: When you CHAIN to another program and you are *not* using the MERGE option, you lose the effect of all DEF-type statements. Therefore it is necessary for you to re-establish variable typing and function definitions immediately after CHAINing. There is a simple way to keep all variables active, to CHAIN to another program, and yet to keep the program's size in memory smaller. This is done by maintaining all variables as integers by default, as shown in this example:

```
10 'FILENAME: "MAIN"
20 DEFINT A-Z
30 DIM X(200), T(50), A$(20), ...
.
.
.
300 CHAIN "SUBPROG1",500,ALL
.
.
.
10 'FILENAME: "SUBPROG1"
.
.
.
500 DEFINT A-Z
.
.
.
```

The CHAIN instruction with the ALL and MERGE options, and the COMMON instruction, immensely improve the flexibility of BASIC as a systems development language. We urge you to begin using these easy-to-use statements so that you can develop your graphics software applications in modular fashion.



CLEAR to Increase Memory

Because the Macintosh has such a rich supply of support software built in, most of the 128K of RAM is taken up with APPLE systems programs, leaving you with only 14K of usable space for your BASIC program. There is a way around that, and it is to use the CLEAR command to steal memory from the Macintosh *Heap*. This area of memory manages windows and desk accessories, so you give up something in order to have more memory for your program. We recommend that you limit your use of the CLEAR instruction to only the largest programs, and only when necessary. Remember that there are other ways to increase memory, such as using integer variables when possible, instead of allowing the system to default to its double-precision variable typing.

The CLEAR command in its simplest form does not save you memory, but it does perform several important actions. If you execute the instruction

50 CLEAR

in your program, BASIC resets *everything*. All files are closed, all COMMON variables are cleared, all numeric and array variables are set to zero, all string variables are set to null, all disk buffers are released to the system, and all stack space, string space, and DEF-typing is reset to the default values. Because the CLEAR is so powerful, you must use it with caution. We recommend that you restrict its use. At first use CLEAR only in the first section of the first program in a series of programs in your largest systems.

When you want to use the CLEAR to increase memory space, remember that the instruction will perform all of the above actions as well. The example below shows a CLEAR instruction that releases 20K bytes from the Macintosh Heap and the desk accessories.

30 CLEAR, 20000

The usual order of operations, which we recommend because we have found that it works well, is shown.

```

10 'FILENAME: "MAIN"
20 CLEAR, 20000
30 DEFINT A-Z
40 DIM B$(20), X(500), T(40), V # (100), ...
.
.
.
300 ON K GOTO 410, 420, 430, 440, ...
.
.
.
410 CHAIN "SUBONE", 500, ALL
420 CHAIN "SUBTWO", 500, ALL
430 CHAIN "SUBTHREE", 500, ALL
440 CHAIN "SUBFOUR", 500, ALL
.
.
.
10 'FILENAME: "SUBONE"
500 DEFINT A-Z
.
.
.

```

(continued)

```
10 'FILENAME: "SUBTWO"  
500 DEFINT A-Z  
.  
.  
.  
10 'FILENAME: "SUBTHREE"  
500 DEFINT A-Z  
.  
.  
.
```



User Interaction with the Mouse

Because the Mouse is such an integral part of the entire Macintosh system, any software that is developed for this machine should make use of this clever device for user input. We have written a simple system to be used on the Macintosh. In so doing we have developed a generalized subroutine that displays a menu of choices, much like the MAC's pulldown menus, and we allow the user to select from the menu with the mouse.

We have coded it within our system as a tool subroutine callable from anywhere in the program. We placed this subroutine that displays the menu and intercepts the mouse drags and clicks in lines 1000 to 1210 of our simple system. You could relocate this module anywhere within your program by RENUMbering the code and saving it as an ASCII file to be MERGED to its new location.

The listing of the program "MENU" below is the driver program that invokes the menu display subroutine. The program's overall function is to display a menu titled "Pfruits". This menu shows three fruits whose names begin with the letter P. When the user makes a selection, the program CHAINs to the appropriate program to display a list of peaches, pears, or plums.

The user can return to the main menu by clicking the mouse anywhere on the screen when within one of these three programs. Notice that line 30 of MENU places the variable D\$ in COMMON for use in all programs in the system. It is defined in line 40 as the filename for this program, "MENU".

Listing, Menu

```
10 ' menu driven system  
20 DIM M$(10)  
30 COMMON D$  
40 D$="MENU"  
50 X=200:Y=45  
60 READ T$,N  
70 FOR I=1 TO N  
80 READ M$(I)
```

```

90 NEXT I
100 DATA Pfruits,4,Pears,Peaches,Plums,Exit
110 GOSUB 1000
120 IF W>0 AND W<N THEN CHAIN M$(W) ELSE STOP
130 IF INKEY$="" THEN 130 ELSE STOP
1000 ' menu
1010 CLS: CALL TEXTFONT(0)
1020 CALL MOVETO(X+3,Y-10): PRINT T$
1030 X8=X+7:Y8=Y-4
1040 M8=0
1050 FOR I8=1 TO N
1060 Y8=Y8+16
1070 CALL MOVETO(X8,Y8): PRINT M$(I8)
1080 IF LEN(M$(I8))>M8 THEN M8=LEN(M$(I8))
1090 NEXT I8
1100 LINE(X-5,Y-4)-(X+10*M8+5,Y+16*N+4),33,B
1110 P8=N-1
1120 IF MOUSE(0)>0 AND X8 >=X AND X8<=X+10*M8 AND
      Y8>=Y AND Y8<=Y+16*N THEN 120 0
1130 LINE(X,Y+P8*16)-(X+M8*10,Y+P8*16+16),30,B
1140 IF MOUSE(0)<>-1 THEN 1120
1150 X8=MOUSE(5): Y8=MOUSE(6)
1160 IF X8<X OR X8>X+10*M8 OR Y8<Y OR Y8>Y+16*N
      THEN 1120
1170 P8=(Y8-Y)/16
1180 LINE(X,Y+P8*16)-(X+M8*10,Y+P8*16+16),33,B
1190 GOTO 1120
1200 W=P8+1
1210 RETURN

```

Annotated Menu Listing

Let's take each one of these lines in the subroutine above to see the complexities of developing a menu driver that interacts with the user with the mouse.

1000 'Show menu. M\$ = array of menu entries, N = number of menu entries, T\$ = menu title, X & Y = coordinates of upper left corner
1010 CLS: CALL TEXTFONT(0) 'Chicago font
1020 CALL MOVETO(X+3,Y-10): PRINT T\$ 'Position cursor and print menu title
1030 X8=X+7: Y8=Y-4 'Indent menu selections. X8 & Y8 are local variables for cursor position
1040 M8=0 'Maximum entry length in characters
1050 FOR I8=1 TO N 'Display entries and get maximum entry length M8

```

1060  Y8 = Y8 + 16 'Adjust down 16 pixels to next line
1070  CALL MOVETO(X8,Y8): PRINT M$(18) 'Move and print
1080  IF LEN(M$(18)) > M8 THEN M8 = LEN(M$(18)) 'Get max. length M8
1090  NEXT I8
1100  LINE(X-5,Y-4)-(X+10*M8+5,Y+16*N+4),33,B 'Draw box around
      entire menu
1110  P8 = N - 1 'P8 points to menu entry. Initial value is assumed to be last entry
1120  IF MOUSE(0) > 0 AND X8 >= X AND X8 <= X + 10*M8 AND Y8 >= Y
      AND Y8 <= Y + 16*N THEN 1200 'Is the mouse within selection area and is
      button up after drag? If so, return W, menu number selected.
1130  LINE(X,Y + P8*16)-(X + M8*10,Y + P8*16 + 16),30,B 'Blank out box
      around entry pointed to but not selected
1140  IF MOUSE(0) < > -1 THEN 1120 'If clicked but not a drag, return to check
      mouse again
1150  X8 = MOUSE(5): Y8 = MOUSE(6) 'Coordinates of drag end where user
      released button
1160  IF X8 < X OR X8 > X + 10*M8 OR Y8 < Y OR Y8 > Y + 16*N THEN 1120 'Not
      within menu area
1170  P8 = (Y8 - Y) \ 16 'User released in menu entry # P8
1180  LINE(X,Y + P8*16)-(X + M8*10,Y + P8*16 + 16),33,B 'Place box around
      menu selection under dragged pointer
1190  GOTO 1120 'Keep checking mouse
1200  W = P8 + 1 'Menu item selected
1210  RETURN

```

Before we proceed with any further annotation of the code, look at a typical main program's menu display, shown in Illustration 7.3.

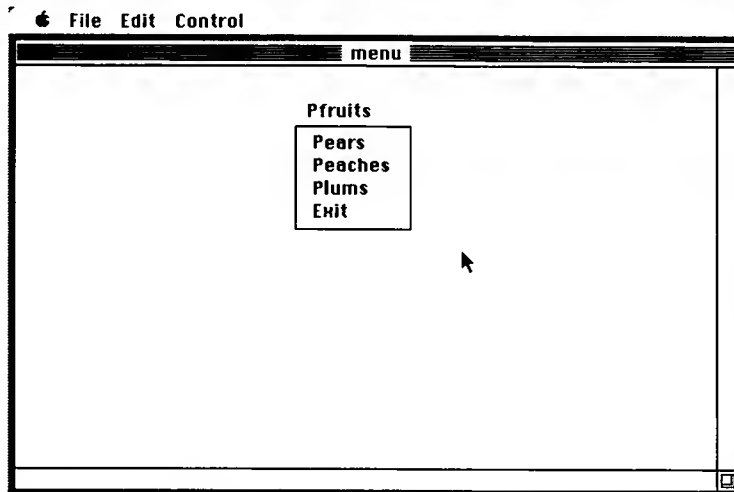


Illustration 7.3 Menu of Pfruits

Illustration 7.4 shows the Pfruits menu display again, only this time the user has stopped the drag on the Plums entry, and a rectangle is drawn around that entry. Illustration 7.5 shows the output from the Plums program.

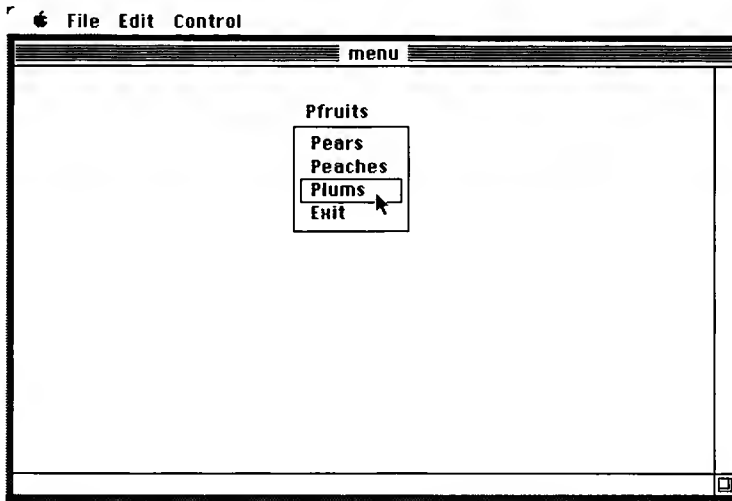


Illustration 7.4 Pfruits menu with Plums selected

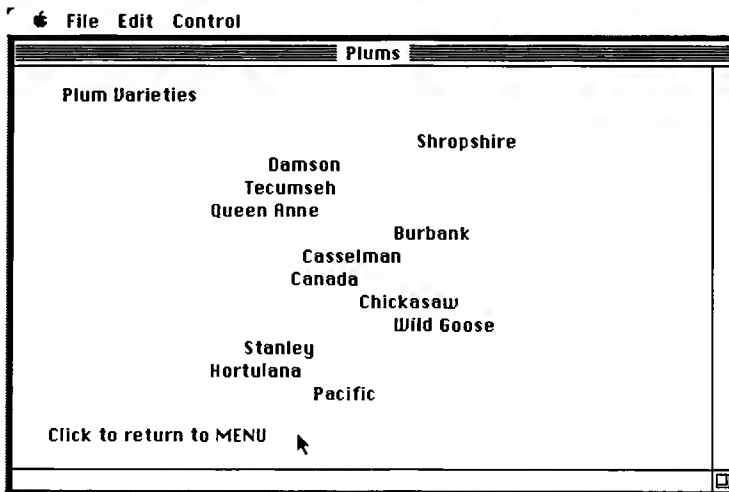


Illustration 7.5 Plums program output

The listings that follow show the three programs that complete the Pfruits program. These are CHAINED to from the main program (MENU). Notice that in each one, the last two lines are:

```
150 CALL MOVETO(25,265): PRINT "Click to return to "+D$
160 IF MOUSE(0)<>0 THEN CHAIN D$ ELSE 160
```

Line 150 moves the cursor position to the bottom of the screen so that the message "Click to return to MENU" doesn't disturb the program's output. The variable D\$ is the one that was placed in COMMON in the first program. It provides the link back to the main menu driver whose filename is "MENU".

Line 160 checks the condition of the mouse. If it is clicked, the program CHAINS to MENU. If not, it continues to check in an infinite loop.

Listing, Pears Program

```
10 ' Pears
20 DIM P$(12),P(12)
30 FOR I=1 TO 12
40 READ P$(I)
50 NEXT I
60 DATA "Anjou","Winter Nelis","Hardy","Bartlett"
70 DATA "Bosc","Wilder Early","Clapp Favorite",
  "Comet"
80 DATA "Comice","Kieffer","Seckel","Early
  Bartlett"
90 CLS
100 PRINT: PRINT TAB(5);"Pear Varieties": PRINT
110 FOR I=1 TO 12
120 P(I)=3*I+10
130 PRINT TAB(P(I));P$(I)
140 NEXT I
150 CALL MOVETO(25,265): PRINT "Click to return to
  "+D$
160 IF MOUSE(0)<>0 THEN CHAIN D$ ELSE 160
```

Listing, Peaches Program

```
10 ' Peaches
20 DIM P$(12),P(12)
30 FOR I=1 TO 12
40 READ P$(I)
50 NEXT I
60 DATA "Carmen","Waddell","Elberta","Rochester"
70 DATA "Champion","Waldo","Wilma","Honey"
```

```

80 DATA "Greensboro", "Cabler", "Chairs", "Fitzgerald"
90 CLS
100 PRINT: PRINT TAB(5); "Peach Varieties": PRINT
110 FOR I=1 TO 12
120 P(I)=35-ABS(13-I-I)
130 PRINT TAB(P(I)); P$(I)
140 NEXT I
150 CALL MOVETO(25,265): PRINT "Click to return to
      "+D$
160 IF MOUSE(0)<>0 THEN CHAIN D$: STOP ELSE 160

```

Listing, Plums Program

```

10 ' Plums
20 DIM P$(12), P(12)
30 FOR I=1 TO 12
40 READ P$(I)
50 NEXT I
60 DATA "Shropshire", "Damson", "Tecumseh",
      "Queen Anne"
70 DATA "Burbank", "Casselman", "Canada", "Chickasaw"
80 DATA "Wild Goose", "Stanley", "Hortulana",
      "Pacific"
90 CLS: RANDOMIZE TIMER
100 PRINT: PRINT TAB(5); "Plum Varieties": PRINT
110 FOR I=1 TO 12
120 P(I)=RND(1)*25+12
130 PRINT TAB(P(I)); P$(I)
140 NEXT I
150 CALL MOVETO(25,265): PRINT "Click to return to
      "+D$
160 IF MDUSE(D)<>D THEN CHAIN D$ ELSE 160

```


PIXEL GRAPHICS AND ICONS

This chapter introduces graphics production with programs written in BASIC. Because Microsoft BASIC has been adopted by so many microcomputer vendors, the programs in the rest of the book tend to be adaptable to many other machines. This is one of the major strengths of Microsoft's version of BASIC. Its graphics commands are powerful and easily used.

What Microsoft does not provide as tools for graphics programming, the Macintosh makes available in its Quickdraw ROM. The 64K Macintosh ROM contains a whole host of prewritten programs. The cluster of programs which Apple refers to as its Quickdraw routines is available through Microsoft BASIC by way of the CALL statement. With the CALL statement, you can write programs that begin to approach the glorious graphics of MacPaint. Because *you* are writing them, however, these programs have great flexibility and can be altered to suit many more applications.

This chapter contains many small programs, illustrating graphics production with BASIC. We introduce several concepts whose understanding is essential to the creation of images on the Macintosh screen.

The simplest and most important component of any graphics image is the pixel, which, as you may remember, is the smallest element of an image on the screen. On the Macintosh, the pixel is a square dot. Macintosh's *resolution* determines how many pixels can fit on the screen. The screen image is made up of 512 vertical columns and 342 horizontal rows forming 174,104 pixels.

Each one of the pixels is either black or white (the Mac is not yet a color machine) and can be represented internally as a single bit, on or off. The entire screen image is stored, this way, as 10,944 16-bit integers. In Chapters 9 and 12, you will see how the integer representation of the screen image can be manipulated to make patterns of your choosing.

Each of the pixel positions on the screen is individually addressable by its column number (X-coordinate) and by its row number (Y-coordinate) as shown in Illustration 8.1. The top left corner's address is (0,0); the top right, (511,0); the bottom left, (0,341); and the bottom right, (511,341). Notice that the X-coordinate comes first, as in Cartesian coordinate geometry. Notice also that the X-coordinates start at the left

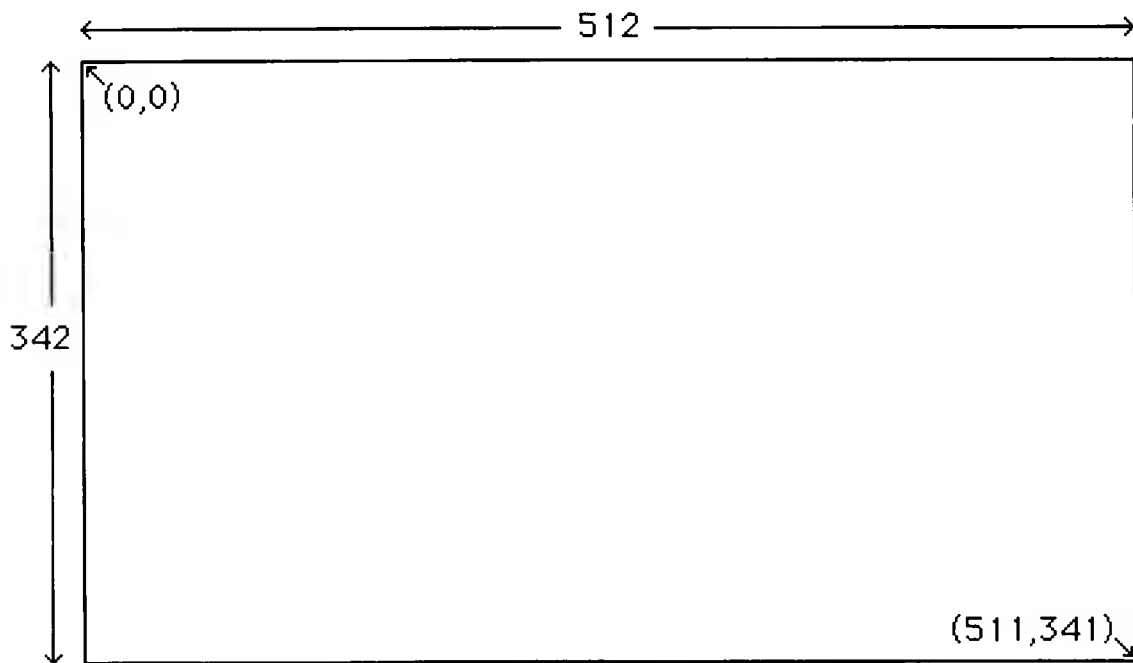


Illustration 8.1 512 x 342 screen grid and coordinate system

edge of the screen at 0 and proceed to the right to 511, whereas the Y-coordinates start at the top of the screen at 0, and go down to 341. This is contrary to the ordinary Cartesian coordinates you see in Illustration 8.2, and is important to remember.

Microsoft BASIC uses four commands to manipulate individual pixels. These are two instructions, PSET and PRESET; and two functions, POINT and PTAB.

The PSET syntax can be in any one of four forms:

1. **PSET (x,y)** draw a black pixel at screen coordinate position x,y.
2. **PSET STEP (dx,dy)** move dx columns and dy rows from the present location and draw pixel.
3. **PSET (x,y,c)** draw pixel with color c at location x,y. The only two colors available on the Macintosh (as of this writing) are black (color = 33) and white (color = 30). Any integer other than these two assumes the color black.
4. **PSET STEP (dx,dy,c)** move dx columns and dy rows, draw pixel with color c.

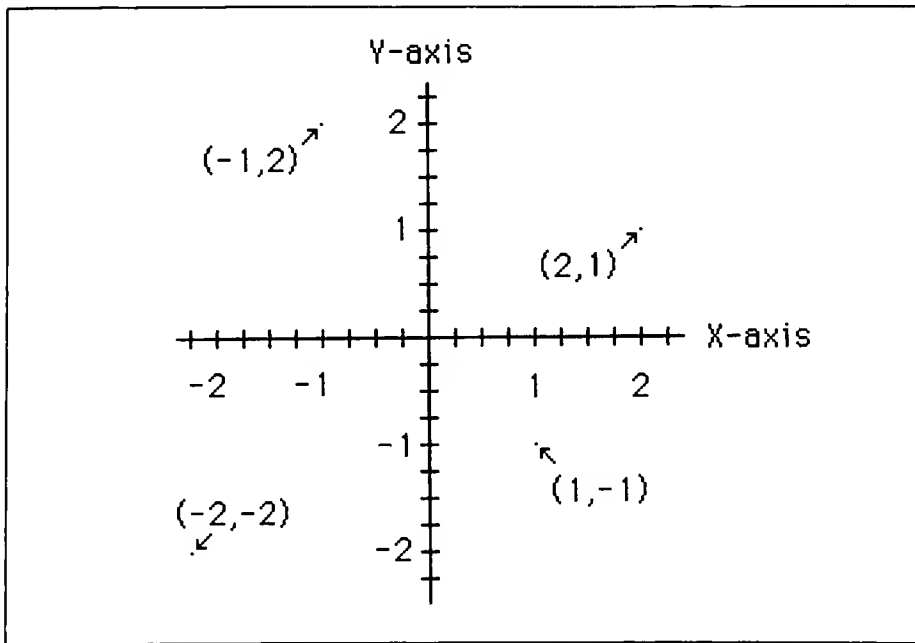


Illustration 8.2 Cartesian coordinate system

The two functions POINT and PTAB are used as follows:

1. **X = POINT(x,y)** determine the color of the pixel at x,y and place it in X. POINT is useful in an IF statement to find out if a specific location has been painted black.
2. **PRINT PTAB(x); v** prints the variable v starting at the horizontal position x. The 512 column positions are all available as starting positions for printing variables, either numeric or string. PRINT PTAB has an interesting feature. If the current print position is to the right of the PTAB pixel position, the variable retreats to the PTAB pixel on the same line. You can write text from right to left using this feature, if you're careful — or weird — or boustrophedonic (look it up).

Application 1: Binomial distribution

This program is called Pachinko after the game of that name which resembles a pinball machine. The program simulates a process that determines the path of a ball through a maze of pins. Consider a large flat board with nails placed into it in a

pattern (Illustration 8.3) with a marble positioned carefully on the topmost nail. When the board is tipped, the marble can take any path around the pins.

Each time the marble lands on a nail, it can go either left or right. As it proceeds through the maze, it will go left on the average about as many times as it goes right. Therefore the most frequently traveled paths are those down the center of the board. The likelihood of a marble always taking a left, or always a right, is very low.

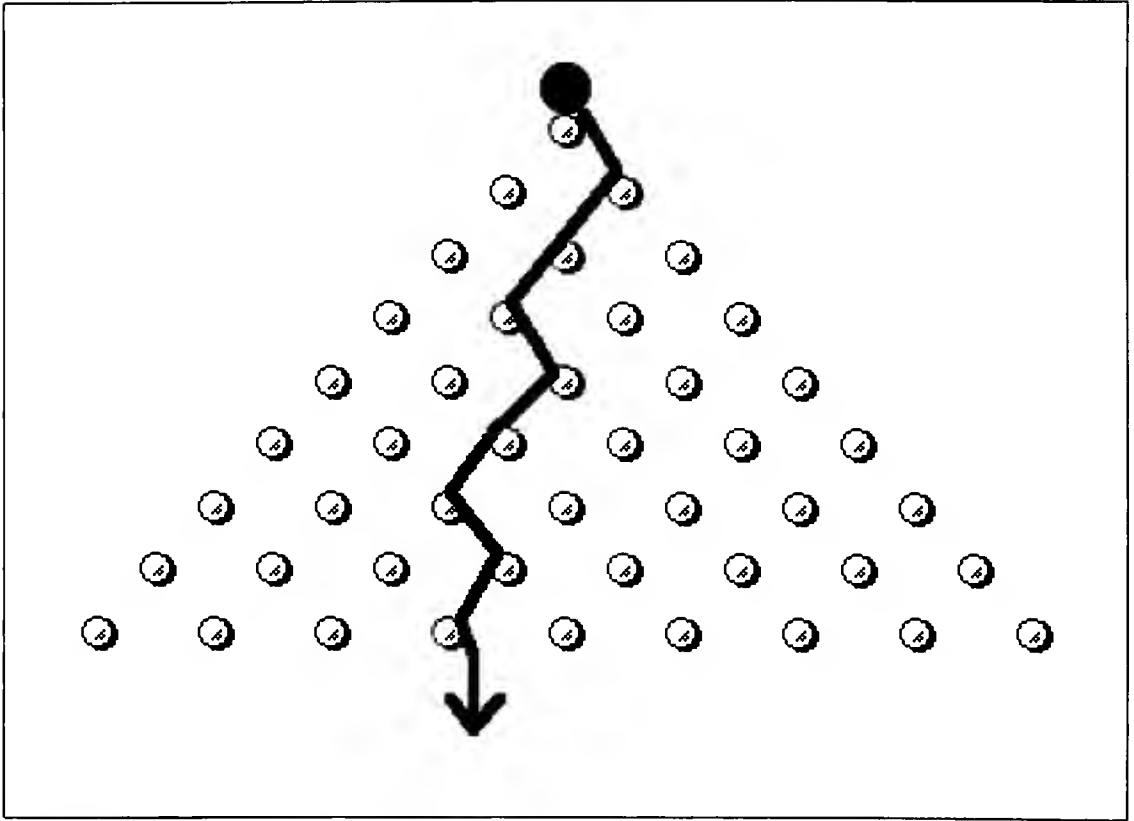


Illustration 8.3 Pachinko board

If you place vertical traps below the rows of pins to trap the marbles as they fall, you can actually observe the distribution of the paths taken. The program simulates this process, by providing columns to hold the marbles as they fall through. Illustrations 8.4 and 8.5 show two stages of output from the program.

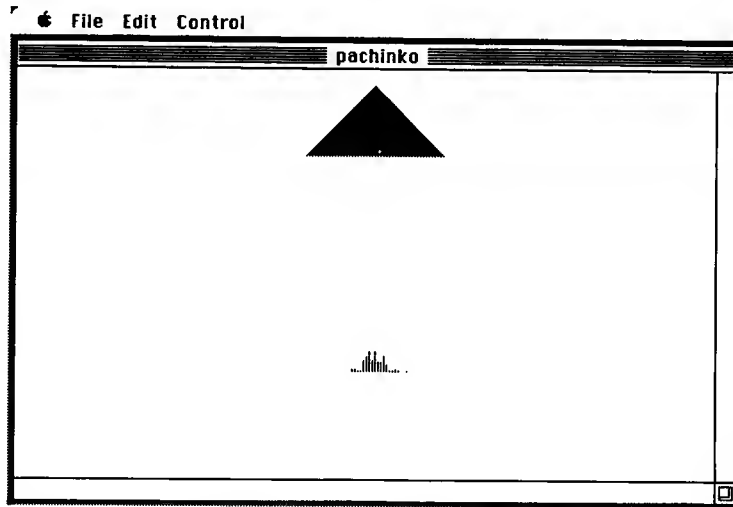


Illustration 8.4 Early output from program "PACHINKO"

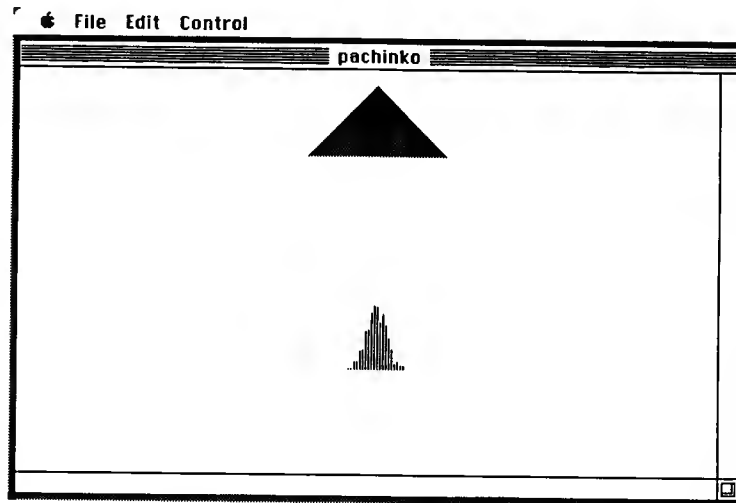


Illustration 8.5 Later output from program "PACHINKO"

Below is a listing of the program.

Listing, PACHINKO

```
10 ' pachinko
20 CLS
30 RANDOMIZE TIMER
40 DIM N(50)
50 X=250:Y=10:L=1:H=49:H2=(H+1)/2
60 B=Y+H+H+100
70 FOR I=1 TO H
80 J=X-L*(I-1)
90 FOR K=1 TO I
100 PSET(J,Y+L*I)
110 J=J+L+L
120 NEXT K
130 NEXT I
140 FOR I=1 TO H+1
150 N(I)=0
160 NEXT I
170 FOR M=1 TO 10000
180 J=X
190 FOR I=1 TO H
200 PRESET(J,Y+L*I)
210 FOR Z=1 TO 5: NEXT Z
220 PSET(J,Y+L*I)
230 IF RND>.5 THEN J=J-L ELSE J=J+L
240 NEXT I
250 W=H2-(J-X)/(L+L)
260 N(W)=N(W)+1
270 PSET(J,B-N(W))
280 NEXT M
```

You can follow the program step by step, outlined here in pseudocode.

- | | | |
|-----|---------|--|
| 1. | 20 | Clear the screen. |
| 2. | 30 | Start the random number generator with a random seed. |
| 3. | 40 | Reserve 50 counters — N. |
| 4. | 50 | Position top of pyramid at X = 250, Y = 10. |
| 5. | | Set L, distance between pins in pyramid to 1. |
| 6. | | Set H, height of pyramid, to 49. |
| 7. | | Calculate H2, midheight of pyramid. |
| 8. | 60 | Set B, base of histogram portion of display, to Y + H + H + 100. |
| 9. | 70-130 | Build pyramid of dots. |
| 10. | 140-160 | Zero all counters N. |
| 11. | 170-280 | Do forever: |
| 12. | 180 | Start X at middle of pyramid. |
| 13. | 190-240 | For I = 1 to H (height of pyramid) do: |
| 14. | 200-210 | White out this pixel, pause. |

- | | | |
|-----|-----|--|
| 15. | 220 | Color this pixel. |
| 16. | 230 | Set direction left or right at random, advance to next row down. |
| 17. | 240 | Enddo. |
| 18. | 250 | Increment appropriate counter. |

Notes:

We have generalized this program purposely so that you can alter it to suit your tastes. Line 50 in particular sets up several important variables. With minor alteration these variables produce quite different displays. The screen image in Illustration 8.6, shows the results of changing L from 1 to 2.

You can increase the height of the pyramid, the spacing between pins, the size of the ball, even the random distribution. You can skew the distribution as if the board were tilting slightly to the right or left by altering line 230. For example:

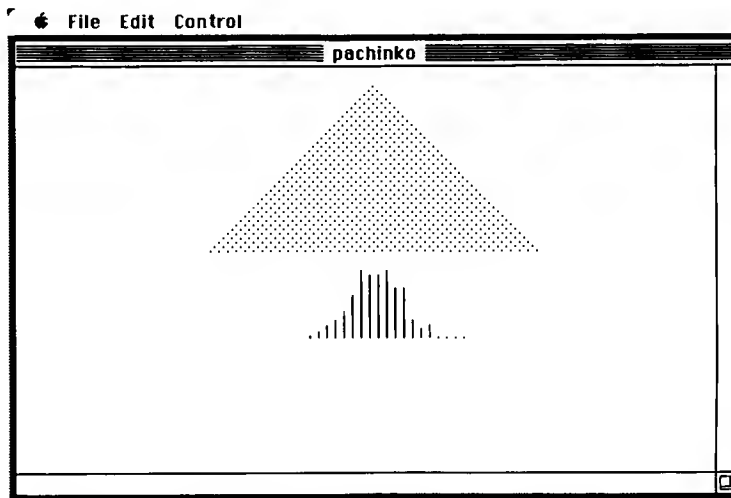


Illustration 8.6 Output, more widely spaced pins

```
IF RND > .3 THEN J = J - L ELSE J = J + L 'Skew left
IF RND > .7 THEN J = J - L ELSE J = J + L 'Skew right
```

This change makes the ball fall to the right or to the left 7 out of 10 times, which produces a strikingly different distribution.

Instead of the random distribution seen in Illustration 8.6, you can produce others by altering line 230. It's a visually captivating way to study chance events.

Application 2: Mathematically Derived Curves

The following series of small programs is based on a common thread of ideas. The theory is to use the PSET instruction to trace a series of points as a mathematical function advances through a series of iterations. The curves and shapes formed are familiar territory to graphics programmers, not only because they are pretty to look at, but because they are useful in many practical applications. Their derivation was made necessary in order to track physical movements.

The examples that follow are indicative of the way a mathematical expression can be translated into a BASIC program. We used various sources such as encyclopedias, dictionaries, and handbooks of mathematical tables. Here are two sources to get you started: *Van Nostrand's Scientific Encyclopedia, 3rd. Edition* (D. Van Nostrand Company, Inc., New Jersey, 1958) and *CRC Standard Mathematical Tables, 14th. Edition* Samuel Selby, Editor (The Chemical Rubber Co., Cleveland, 1965).

We tested various values for variables until we were satisfied with the end result. One of the great rewards of programming these mathematical functions was being able to see from the book what the end result should look like on the screen. We were surprised in some cases, and learned much in the process.

In describing each of the following curves, we first provide you with a sketch of the curve as described in the references we mention above. At the bottom of the sketch we include the mathematical description of the curve, followed by the actual output that the Macintosh produces. Finally, we show you the listing that produced the output.

Prolate Cycloid

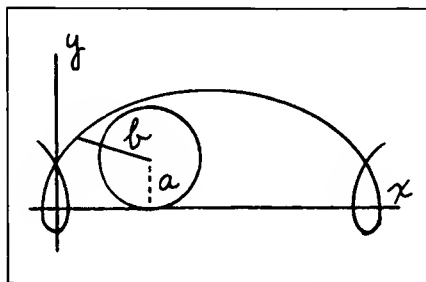


Illustration 8.7 Prolate cycloid in
Cartesian coordinates

$$x = a\theta - b \sin \theta, \quad y = a - b \cos \theta, \quad a < b$$

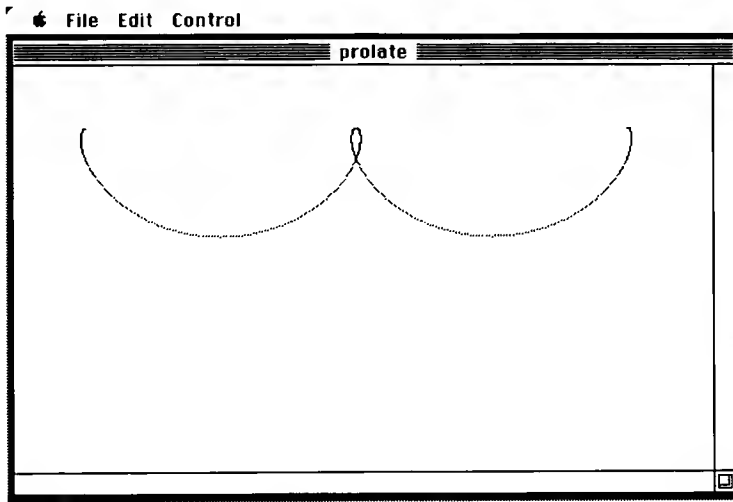


Illustration 8.8 Macintosh output, prolate cycloid

The output (Illustration 8.8) is upside down from Illustration 8.7, which shows the usual orientation found in the math books. The reason for this is that the Y-coordinate on the Macintosh goes positive *down* the screen. This is opposite of its positive direction in the Cartesian system.

To re-orient the computer's output to match the format shown in most math books, simply change line 90 in the listing of "Prolate Cycloid" to:

90 $Y = B * \cos(R) - A$

Listing, Prolate Cycloid

```

10 ' Prolate cycloid
20 CLS
30 C=50:D=50:A=30
40 FOURPI=16*ATN(1)
50 PIOVER100=FOURPI/400
60 B=A*1.25
70 FOR R=0 TO FOURPI STEP PIOVER100
80 X=A*R-B*SIN(R)
90 Y=A-B*COS(R)
100 PSET(C+X,D+Y)
110 NEXT R

```

Suggestions:

```
40 FOR D=50 TO 200 STEP SQR(D)
120 NEXT D
```

```
60 FOR B=.75*A TO 2*A STEP .2
120 NEXT B
```

Curtate Cycloid

Change the Prolate Cycloid, which has closed loops at every cycle, to the Curtate Cycloid, which has no loops, (Illustration 8.9) by keeping $A > B$.

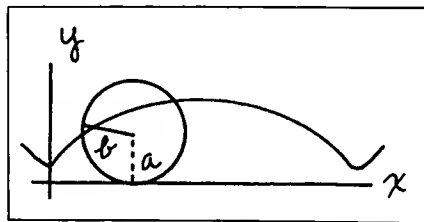


Illustration 8.9 Curtate cycloid,
Cartesian coordinates

$$x = a\theta - b \sin \theta, \quad y = a - b \cos \theta, \quad a > b$$

Involute of Circle

Notice that again the curve doesn't go in the same direction as on the sketch, for the same reason that the cycloids were upside down.

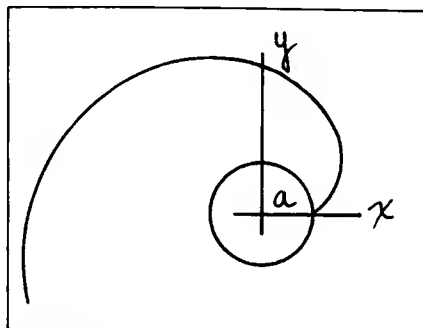


Illustration 8.10 Involute of circle

$$x = r \cos \theta + r\theta \sin \theta, \quad y = r \sin \theta - r\theta \cos \theta$$

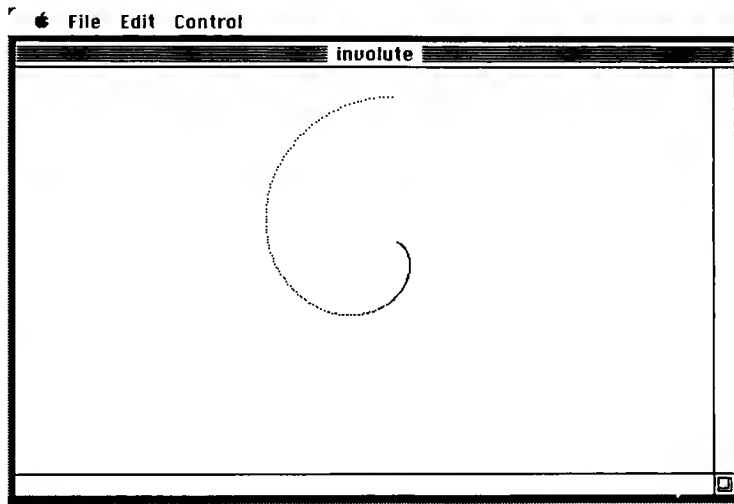


Illustration 8.11 Macintosh output, Involute of circle

Listing, Involute of circle

```

10 ' Involute of Circle
20 CLS
30 C=250:D=120:A=16
40 TWOPI=8*ATN(1)
50 PIOVER100=TWOPI/200
60 FOR R=0 TO TWOPI STEP PIOVER100
70 X=A*COS(R)+A*R*SIN(R)
80 Y=A*SIN(R)-A*R*COS(R)
90 PSET(C+X,D+Y)
100 NEXT R

```

Suggestions:

```

55 FOR A=8 TO 24 STEP 2
110 NEXT A

```

85 Y=-Y 'this reverses the direction of the curve

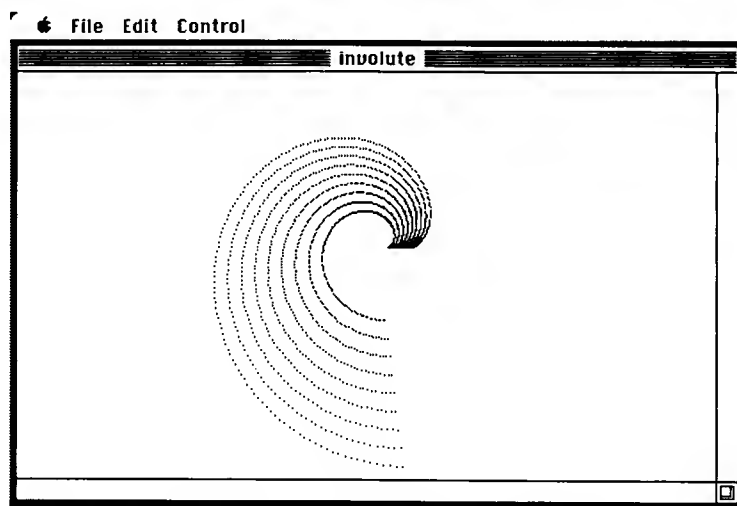


Illustration 8.12 Macintosh output of multiple involutes

Cardioid

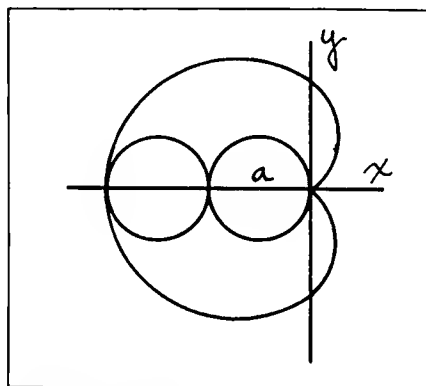


Illustration 8.13 Cardioid

$$(x^2 + y^2 + ax)^2 = a^2(x^2 + y^2),$$

$$p = a(1 - \cos \theta) \quad \text{or} \quad p = -a(1 + \cos \theta)$$

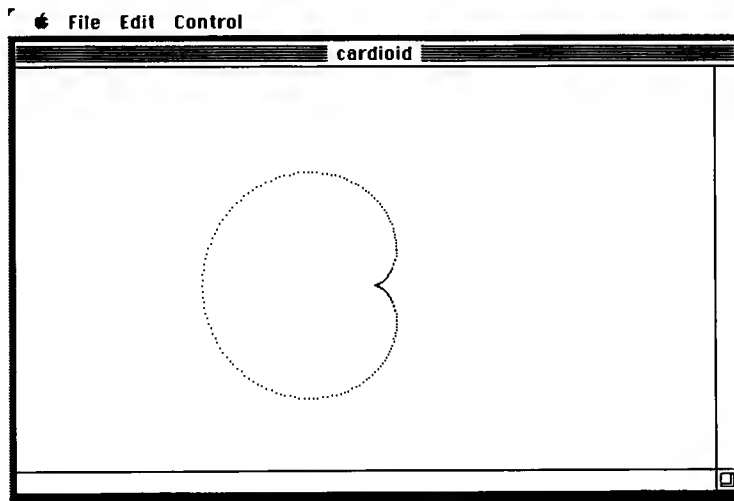


Illustration 8.14 Macintosh output, Cardioid

Listing, Cardioid

```

10 ' Cardioid
20 CLS
30 C=250:D=150:A=60
40 TWOPI=8*ATN(1)
50 PIOVER100=TWOPI/200
60 FOR R=0 TO TWOPI STEP PIOVER100
70 T=A*(1-COS(R))
80 X=T*COS(R)
90 Y=T*SIN(R)
100 PSET(C+X,D+Y)
110 NEXT R

```

Suggestions:

```

40 FOR A=10 TO 100 STEP 10 'or step sqrt(a)
120 NEXT A

```

Reverse the heart's direction with

```

100 PSET(C-X,D-Y)
55 FOR C=200 TO 500 STEP 20
57 D=(C-50)/2
120 NEXT C

```

Evolute of Ellipse

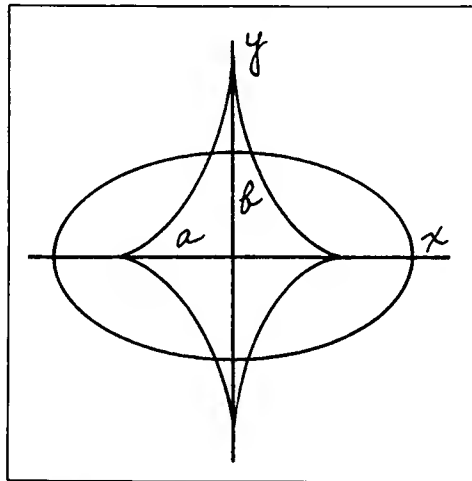


Illustration 8.15 Evolute of Ellipse

$$x = A \cos^3 \theta, \quad y = B \sin^3 \theta$$

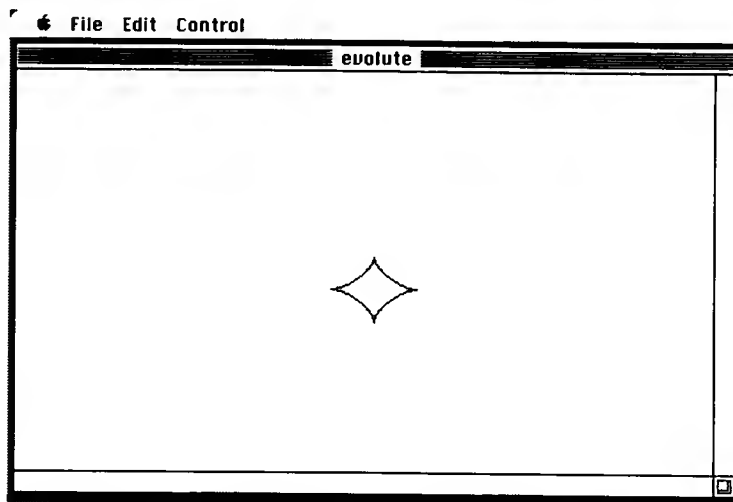


Illustration 8.16 Macintosh output, Evolute of ellipse

Listing, Evolute of Ellipse

```

10 ' Evolute of Ellipse
20 CLS
30 C=250:D=150:A=30
40 TWOPI=8*ATN(1)
50 PIOVER100=TWOPI/200
60 B=A*.75
70 FOR R=0 TO TWOPI STEP PIOVER100
80 X=A*COS(R) 3
90 Y=B*SIN(R) 3
100 PSET(C+X,D+Y)
110 NEXT R

```

Suggestions: As usual, form a loop changing the angle A. Or play with the derivations of X and Y, such as $X=A*\cos(R)^2$. You can flatten the image by changing line 60 to $B=A*.4$

Hypocycloid of Four Cusps (Astroid)

Note that the word is Astroid, not Asteroid.

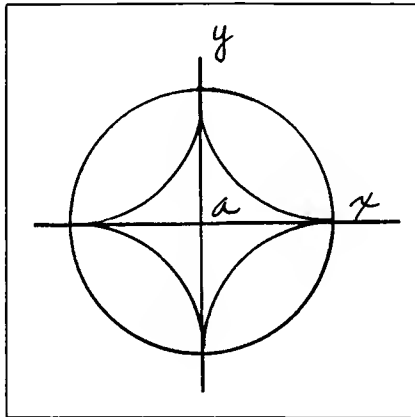


Illustration 8.17 Hypocycloid of Four Cusps (Astroid)

$$x = a \cos^3 \theta, \quad y = a \sin^3 \theta$$

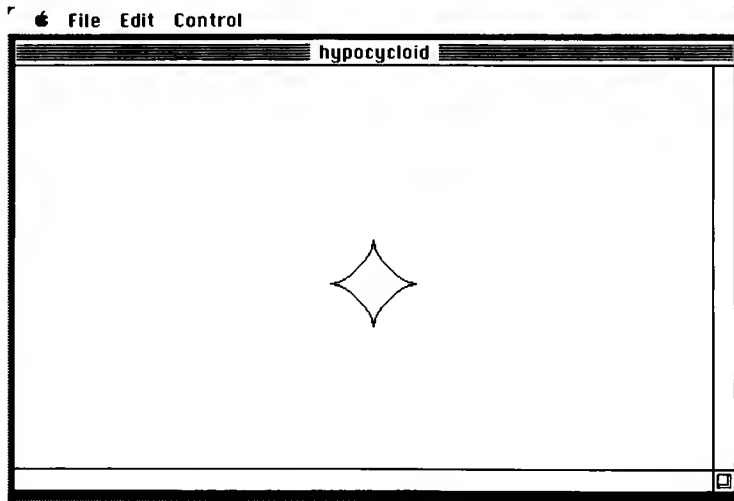


Illustration 8.18 Macintosh output, Hypocycloid

This image is closely related to the Evolute of Ellipse shown previously. This one is based on a circle, while the Evolute is based on an ellipse.

Listing, Hypocycloid of Four Cusps (Astroid)

```
10 ' Hypocycloid of Four Cusps (Astroid)
20 CLS
30 C=250:D=150:A=30
40 TWOPI=8*ATN(1)
50 PIOVER100=TWOPI/200
60 FOR R=0 TO TWOPI STEP PIOVER100
70 X=A*COS(R) 3
80 Y=A*SIN(R) 3
90 PSET(C+X,D+Y)
100 NEXT R
```

Roses

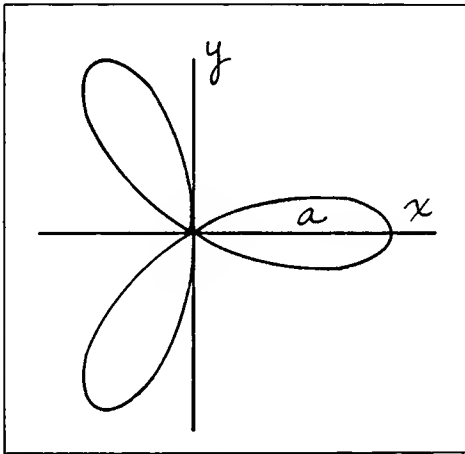


Illustration 8.19 Three-leaved rose

$$p = a \cos 3\theta$$

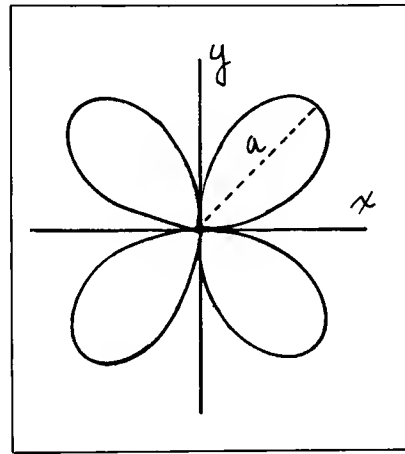


Illustration 8.20 Four-leaved rose

$$p = a \sin 2\theta$$

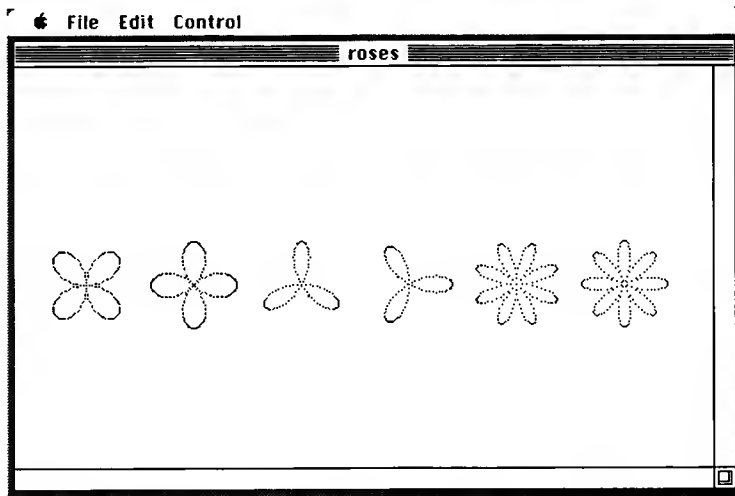


Illustration 8.21 Various Macintosh roses

Listing, Roses

```
10 ' Roses
20 CLS
30 C=50:D=150:A=30
40 TWOPI=8*ATN(1)
50 PIOVER100=TWOPI/200
60 S=0
70 FOR I=3 TO 8
80 N=INT((I+1)/2)
90 S=1-S
100 FOR R=0 TO TWOPI STEP PIOVER100
110 IF S=0 THEN T=A*COS(N*R) ELSE T=A*SIN(N*R)
120 X=T*COS(R)
130 Y=T*SIN(R)
140 PSET(C+X,D+Y)
150 NEXT R
160 C=C+75
170 NEXT I
```

Notes: This program produces enough roses that you can begin to understand what's going on. The little table below shows the values of I, N, and S for all six images.

I	N	S	T
3	2	1	$A \sin(nr)$
4	2	0	$A \cos(nr)$
5	3	1	$A \sin(nr)$
6	3	0	$A \cos(nr)$
7	4	1	$A \sin(nr)$
8	4	0	$A \cos(nr)$

Illustration 8.22 Table of relationships among roses

To increase the number of lobes, increase N; but note that when N is 2, the two lobes are duplicated across the center, making it look like four lobes. If you set n to 5, you get five lobes; but if you set N to 6, you get 12 lobes. To orient the rose with one lobe vertical, set S to 0.

We can't leave this topic without providing you with a hint at some other curves so that you can try a few exercises of your own. We have imbedded within phrases the names of some of our favorite mathematically derived curves for you to program. Their names prove that some mathematicians are poets at heart. Here we suggest some jargon you can take to your next party to impress your friends.

"Waiter! I have a Limacon of Pascal in my soup!"

"Why don't you dress up as a Witch of Cassini for the party?"

"Come on over sometime, and I'll show you my Cissoid of Diocles"

"My Strophoid and Ovals of Cassini need work, but what a Bifolium!"

"Your tie resembles Bernoulli's Lemniscate."

"Folium of Descartes leads Conchoid of Nicomedes by a nose, and Spiral of Archimedes trails the pack."

And so on.

Application 3: Birthdays

The next time you get together with 25 or more people, find out if two people were born on the same day and month. If you had a gang of 182 people you might think that the odds of two people having the same birthday would be about fifty-fifty (182 is half of 365). In fact, the odds are much better than that. For a full discussion of this probability problem, we refer you to *Popular Computing*, July 1984, p. 190. The program we include here is an adaptation of the one in that magazine; we even kept some of the variables.

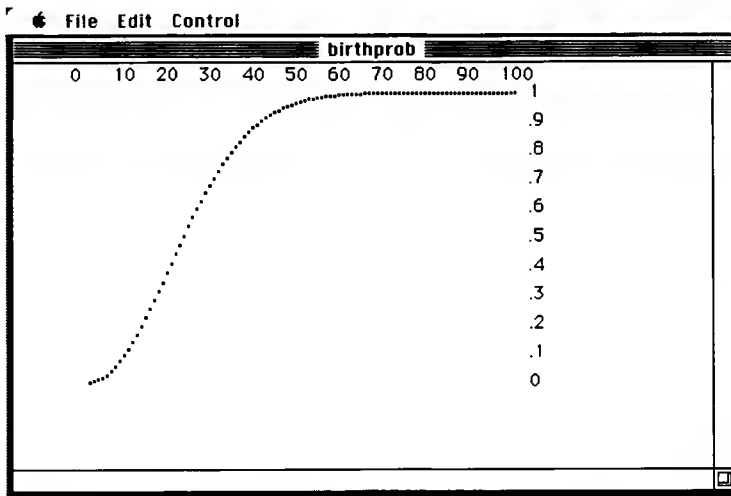


Illustration 8.23 Macintosh output, Birthday program

Listing, Birthday

```
10 ' birthday
20 ' adapted from Popular Computing, July 1984, p 190.
30 DIM X(100)
40 M=100
50 PB=1 'initial probability multiplier
60 FOR G=1 TO M
70 P=(365-G+1)/365: PB=P*PB: X(G)=1-PB
80 NEXT G
90 CLS
```

(continued)

```
100 FOR I=50 TO 350 STEP 30
110 PRINT PTAB(I-18); (I-50)/3;: NEXT I: PRINT
120 FOR J=20 TO 220 STEP 20
130 CALL MOVETO(352,J+4): PRINT (240-J)/200-.1
140 NEXT J
150 CALL TEXTMODE(1)
160 CALL PENSIZE(2,2)
170 FOR G=1 TO M
180 X=G*3+50
190 Y=220-200*X(G)
200 CALL MOVETO(X,Y):PSET(X,Y)
210 NEXT G
999 GOTO 999
```

Below is an outline of the steps in Pseudocode.

30	Save 100 positions of X, probabilities for up to 100 people.
60-80	For each of the 100 populations, calculate the probability.
90	Clear the screen.
100-110	Set horizontal screen positions I from 50 to 350 in steps of 30. Tab to pixel i-18 and print values 0 to 100 in steps of 10.
120-140	For J=20 to 220 in steps of 20 do: Move to column 352, row J+4 Print probability (1 to 0 in steps of .1) Enddo.
150	Set Macintosh textmode to OR what is on the screen, so that it shows through.
160	Set Macintosh pensize to a 2x2 (4-pixel) dot instead of the standard 1x1 (single-pixel) dot.
170-210	For G=1 to 100 do: Calculate X displacement as $3 * G + 350$ Calculate Y displacement as $220 - 200 * X(G)$ Move to location X,Y and place the 2x2 dot there. Enddo.

Notes: This program uses several CALLs to the Macintosh Quickdraw routines, and they need some attention.

CALL MOVETO(x,y) Lines 130 and 200 use a Toolbox subroutine to move the cursor position to a screen pixel location. The two variables (x,y) are absolute screen coordinates.

CALL TEXTMODE(m) In line 150, we reset the usual textmode. This allows us to write without obliterating what is already on the screen. In the listing shown, this line is not necessary. However, you could change line 200 to use a PRINT instead of a PSET if you wanted the chart to be a series of asterisks. Then you would want the last few PRINT statements to stay clear of the "I" printed in the probability column. The PRINT statement carries with it a line width of 15 pixels and one or more blanks at the end of the line. This is the result of the default textmode(0), which causes the

text to replace whatever is on the screen. Mode 1 causes the text output to be superimposed (ORed) on the screen. Here's a summary of the four textmodes you can use.

CALL TEXTMODE(m)

Effect

m = 0 (default)

Text replaces whatever is on screen; copy mode.

m = 1

Text superimposes (ORs) screen image.

m = 2

Screen inverts if pixel exists in text (XOR).

m = 3 (BIC mode)

Screen inverts if pixel is black (Black Is Changed).

CALL PENSIZE(w,h) redefines the pen's dimensions. Line 160 uses this routine to change the PSET's output from one pixel to four, for higher clarity. This CALL is most commonly used when you want to draw thicker lines than the standard (default) one-pixel width. Here we use it to make fatter points.

Application 4: Stars and Motion

Our aim is to show you how to design objects and manipulate them on the screen to simulate motion using the GET and PUT.

The GET and PUT are related graphics instructions that grab (GET) a rectangle of pixels from the screen and place (PUT) them elsewhere, perhaps in a different size. You must define the rectangle as an integer array that can contain as many bits as there are pixels in the image. The BASIC manual's description of how to compute the size of the array to be DIMensioned is a bit complex, so here's a simpler rule: Consider your rectangle to be X by Y pixels. The number of integers you must reserve is $2 + X*Y/16$.

For example, suppose you have a screen image that can be enclosed completely by a rectangle that is 40 pixels across and 35 pixels deep. If you want to GET and PUT the array IMAGE, that array variable must be DIMensioned at least $2 + 40*35/16 = 90$. Therefore the DIM statement would be

```
20 DIM IMAGE(90)
```

There is no way to be wrong if your DIM statement allows for a larger array, such as DIM IMAGE(100) in the above example. However, you *must* reserve at least as much space, in integers, as indicated by the formula $2 + X*Y/16$.

The syntax of the GET and PUT are explained well in the Microsoft BASIC manual, so we won't repeat this. We will show you what we did with these instructions in our programs, and in this way you should discover some good graphics tricks, as we did.

Shooting Star

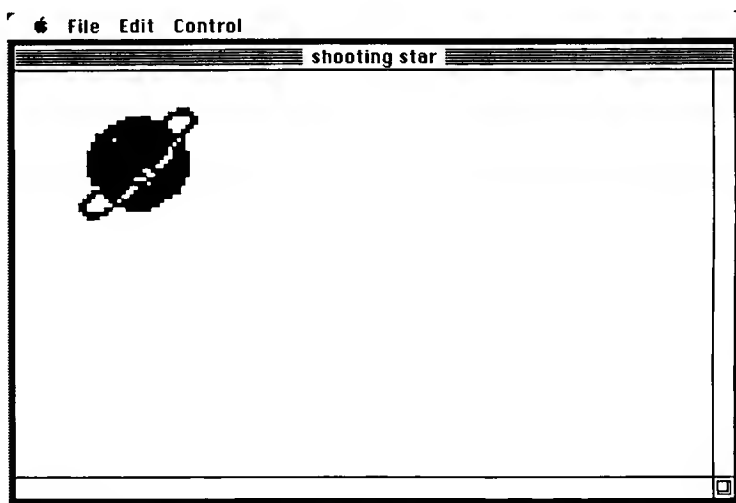


Illustration 8.24 Shooting star at start of run

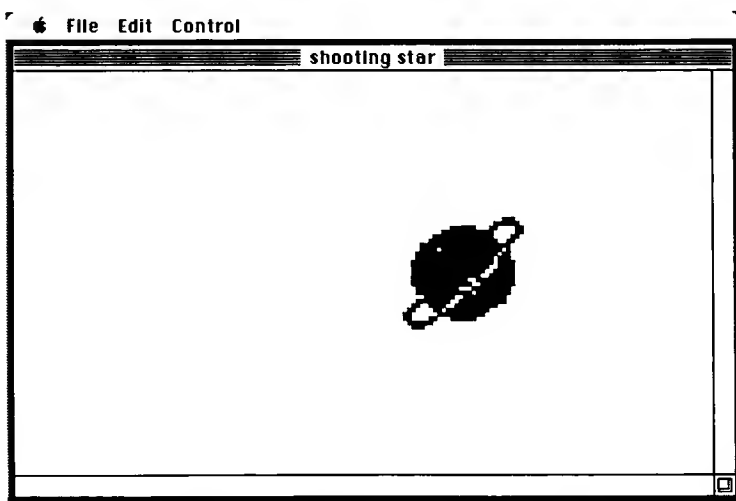


Illustration 8.25 Shooting star halfway through run

Listing, Shooting Star

```

10 ' shooting star
20 DEFINT A-Z
30 DIM A(65),G(5)
40 CLS:INPUT "enter size of star .. ",S
50 CLS
60 X1=5:Y1=5:X2=X1+S*32-1:Y2=Y1+S*32-1
70 A(0)=32:A(1)=32
80 A(2)=0:A(3)=0
90 A(4)=0:A(5)=0
100 A(6)=0:A(7)=0
110 A(8)=0:A(9)=%H70
120 A(10)=0:A(11)=%H1F8
130 A(12)=7:A(13)=%HE30C
140 A(14)=%H1F:A(15)=%HFE0C
150 A(16)=%H7F:A(17)=%HFE18
160 A(18)=%HFF:A(19)=%HFF30
170 A(20)=%H3FF:A(21)=%HFFE0
180 A(22)=%H3DF:A(23)=%HFFA0
190 A(24)=%H7FF:A(25)=%HFF60
200 A(26)=%H7FF:A(27)=%HFEE0
210 A(28)=%HFFF:A(29)=%HFEF0
220 A(30)=%HFFF:A(31)=%HFCF0
230 A(32)=%HFFF:A(33)=%HF9F0
240 A(34)=%HFFF:A(35)=%HF3F0
250 A(36)=%HFFF:A(37)=%H9FF0
260 A(38)=%HFFF:A(39)=%HCF0
270 A(40)=%H7FE:A(41)=%H3FE0
280 A(42)=%H7FF:A(43)=%HDFE0
290 A(44)=%H1FC:A(45)=%HFFC0
300 A(46)=%H7F9:A(47)=%HFFC0
310 A(48)=%HCF3:A(49)=%HFF00
320 A(50)=%H186F:A(51)=%HFC00
330 A(52)=%H303F:A(53)=%HF800
340 A(54)=%H30C7:A(55)=%HE000
350 A(56)=%H1F80:A(57)=0
360 A(58)=%HE00:A(59)=0
370 A(60)=0:A(61)=0
380 A(62)=0:A(63)=0
390 A(64)=0:A(65)=0
400 F=150
410 D=3:E=1
420 FOR I=1 TO F
430 PUT(X1,Y1)-(X2,Y2),A,PSET
440 X2=X2+D:X1=X1+D
450 Y2=Y2+E:Y1=Y1+E
460 NEXT I
470 IF INKEY$="" THEN 470 ELSE 40

```


Below is an outline of the Shooting Star Program.

- 20 Define all variables integer
- 30 Declare Array A for image, reserve 65 integers for it.
- 40 Clear screen, get scale S from user. Scale should be 1-10.
- 50 Clear screen again to remove dialog.
- 60 Define upper left (X1,Y1) and lower right (X2,Y2) coordinates for
rectangle to be manipulated in GET and PUT.
- 70-390 Describe the image to be manipulated and transfer it to the array
A. Because A is DIMensioned 65, a total of 66 16-bit integers
must be defined. The first two positions of the array are always
the size of the array in bits, so A(0) contains the width, of the array
in pixels, and A(1) contains the height of the array, in pixels. Line
60 has described a rectangle that is 32 by 32 for a scale S of 1, 64
by 64 if S=2, and so on. We need only define 32x32 bits, or 64
16-bit integers. Scaling will be taken care of with the PUT.
Because each pair of array elements defines 32 bits, A(2) and
A(3) define the first row of 32 bits; A(4) and A(5) define the
second row; and so on, until A(64) and A(65) define the bottom
row of the 32x32-bit square. The easiest way to build your image
is to take grid paper, sketch out your image, and place a 1 where
there is black, a 0 where the image is white. The result is then
reduced to a rectangle (try to make that rectangle's width divisi-
ble by 16) and translated into hexadecimal from this binary
picture.

For example, suppose you want your image to be a frog, like the lower-case "c" in Cairo font, and you want that image to be contained within a rectangle 16 bits wide by 32 bits deep. First, DIMension the integer array FROG%(33). Define FROG%(0) = 16 and FROG%(1) = 32, the design's column and row dimensions. Then, lay out your pattern in 1s and 0s on grid paper, as shown in Illustration 8.26.

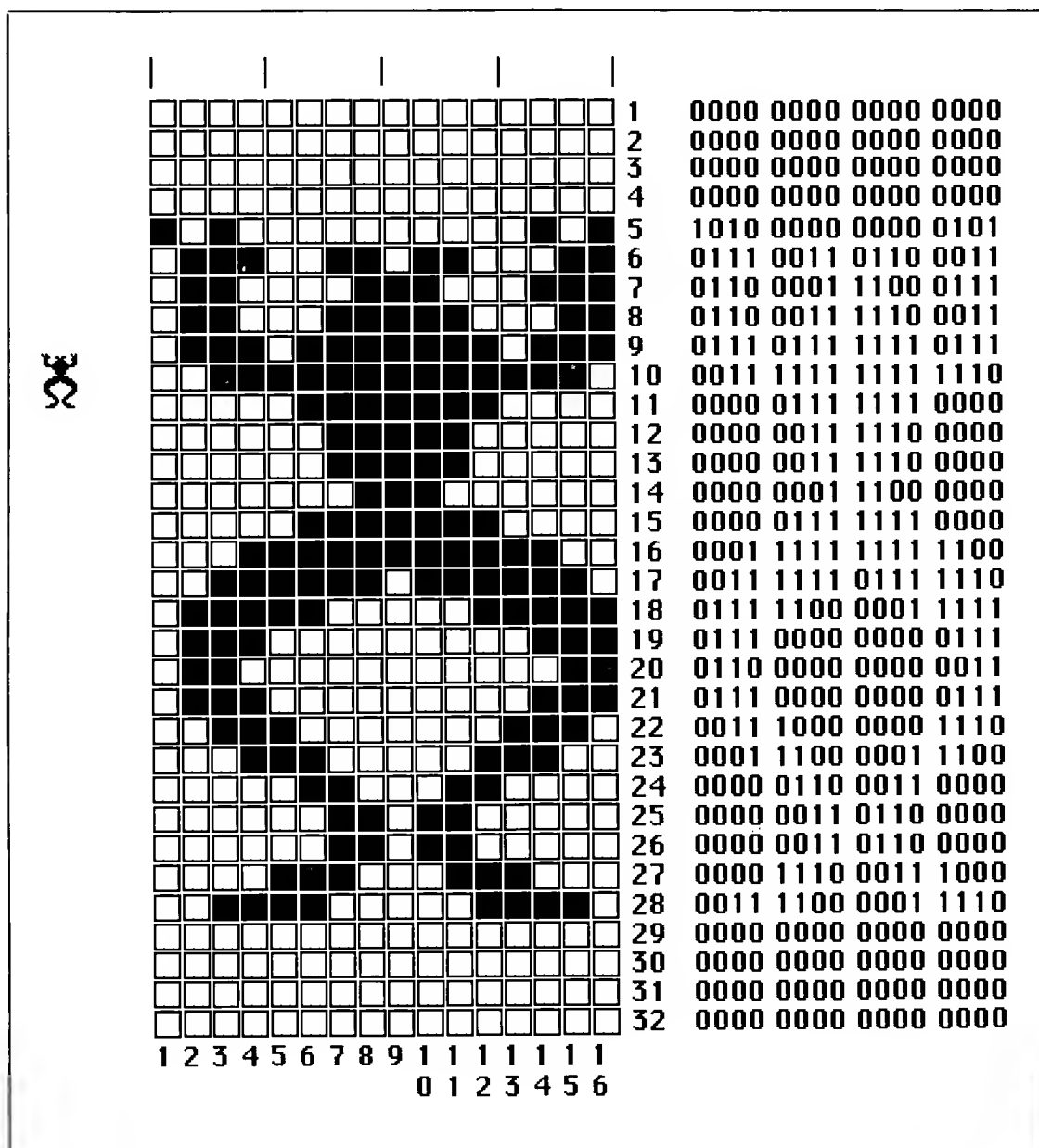


Illustration 8.26 Sketch of 16x32 frog with binary equivalent

Your next task is to transfer that image line by line into the array FROG%, starting with FROG%(2) through FROG%(33). The easiest way to transfer the 16-bit groups is to define them as four hexadecimal digits. For the sake of completeness, we review those for you here in Illustration 8.27.

Binary	Hex	Binary	Hex
0000	0	1000	8
0001	1	1001	9
0010	2	1010	A
0011	3	1011	B
0100	4	1100	C
0101	5	1101	D
0110	6	1110	E
0111	7	1111	F

Illustration 8.27 Bit representation of hexadecimal digits

So the frog's definition is

```

FROG%(0) = 16: FROG%(1) = 32: FROG%(2) = &H0000
FROG%(3) = &H0000: FROG%(4) = &H0000
FROG%(5) = &H0000: FROG%(6) = &HA005
FROG%(7) = &H7363: FROG%(8) = &H61C7
FROG%(9) = &H63E3: FROG%(10) = &H77F7
FROG%(11) = &H3FFE: FROG%(12) = &H07F0
FROG%(13) = &H03E0: FROG%(14) = &H03E0
FROG%(15) = &H01C0: FROG%(16) = &H07F0
FROG%(17) = &H1FFC: FROG%(18) = &H3F7E
FROG%(19) = &H7C1F: FROG%(20) = &H7007
FROG%(21) = &H6003: FROG%(22) = &H7007
FROG%(23) = &H380E: FROG%(24) = &H1C1C
FROG%(25) = &H0630: FROG%(26) = &H0360
FROG%(27) = &H0360: FROG%(28) = &H0E38
FROG%(29) = &H3C1E: FROG%(30) = &H0000
FROG%(31) = &H0000: FROG%(32) = &H0000
FROG%(33) = &H0000

```

Now let's get back to our program.

```

400          Define F, number of times through loop to move the object.
410          Define horizontal displacement D, and vertical displacement E.
420-460      For I = 1 to F DO:
430          PUT the array A into area defined by (X1,Y1) and (X2,Y2). Use
              the PSET action verb to place it there.
440-450      Increment coordinates by D and E.
460          ENDDO.
470          Wait for a keystroke from user, then return to beginning.

```

The reason that two PUTs with the XOR option (as suggested in the manual to simulate motion) aren't needed in this program is because the increments for the picture's shift D and E cause the picture to move so little that the pixels that are turned on aren't left on. This automatic erasure is caused by the picture's definition with a 3-pixel white border surrounding the entire design of the star. If D and E are defined to be greater than three pixels, then some residue is left behind because the new image's border doesn't overlap the old image entirely. Play around with the values of D and E, and you will see what happens.

Enlarge Star program

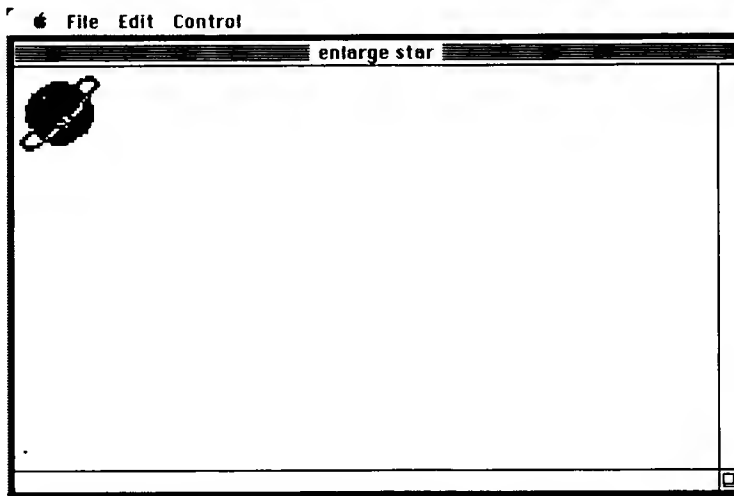


Illustration 8.28 Enlarge Star output when star is small

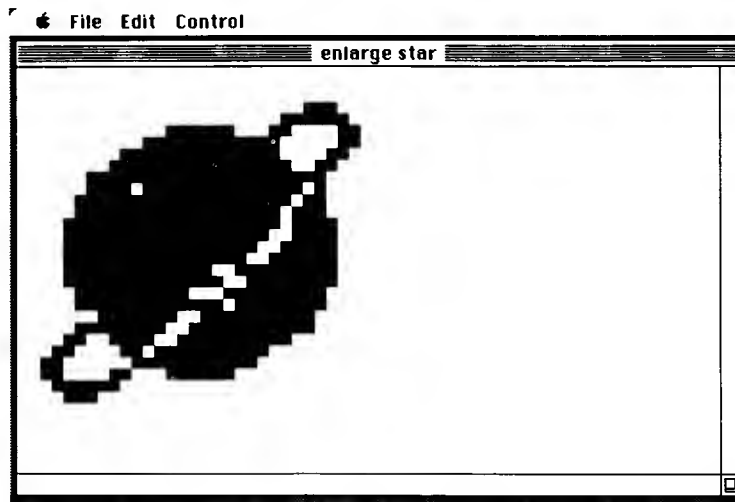


Illustration 8.29 Enlarge Star output when star is large

This program is identical to the last, except we have no loop for moving the star across the screen. What we wanted to show here was the effect of changing the scale S . As you enter different values for S in line 40, the image that is PUT onto the screen changes. When $S = 1$, it is at its real size, as defined in the array. When $S = 2$, the image is twice as large (the area of the square is really four times as much, but each dimension across and down is only twice as much).

Listing, Enlarge Star

```

10 ' enlarge star
20 DEFINT A-Z
30 DIM A(65)
40 CLS:INPUT "enter size of star .. ",S
50 CLS
60 X1=0:Y1=0:X2=X1+S*32-1:Y2=Y1+S*32-1
70 A(0)=32:A(1)=32
80 A(2)=0:A(3)=0
90 A(4)=0:A(5)=0
100 A(6)=0:A(7)=0
110 A(8)=0:A(9)=&H70
120 A(10)=0:A(11)=&H1F8
130 A(12)=7:A(13)=&HE30C
140 A(14)=&H1F:A(15)=&HFE0C
150 A(16)=&H7F:A(17)=&HFE18
160 A(18)=&HFF:A(19)=&HFF30
170 A(20)=&H3FF:A(21)=&HFFE0
180 A(22)=&H30F:A(23)=&HFFA0
190 A(24)=&H7FF:A(25)=&HFF60
200 A(26)=&H7FF:A(27)=&HFEED

```

```

210 A(28)=&HFFF:A(29)=&HFEF0
220 A(30)=&HFFF:A(31)=&HFCF0
230 A(32)=&HFFF:A(33)=&HF9F0
240 A(34)=&HFFF:A(35)=&HF3F0
250 A(36)=&HFFF:A(37)=&H9FF0
260 A(38)=&HFFF:A(39)=&HCFF0
270 A(40)=&H7FE:A(41)=&H3FE0
280 A(42)=&H7FF:A(43)=&HDFF0
290 A(44)=&H1FC:A(45)=&HFFC0
300 A(46)=&H7F9:A(47)=&HFFC0
310 A(48)=&HCF3:A(49)=&HFF00
320 A(50)=&H186F:A(51)=&HFC00
330 A(52)=&H303F:A(53)=&HF800
340 A(54)=&H30C7:A(55)=&HE000
350 A(56)=&H1F80:A(57)=0
360 A(58)=&HE00:A(59)=0
370 A(60)=0:A(61)=0
380 A(62)=0:A(63)=0
390 A(64)=0:A(65)=0
400 PUT(X1,Y1)-(X2,Y2),A
410 IF INKEY$="" THEN 410 ELSE 40

```

Racing Stars program

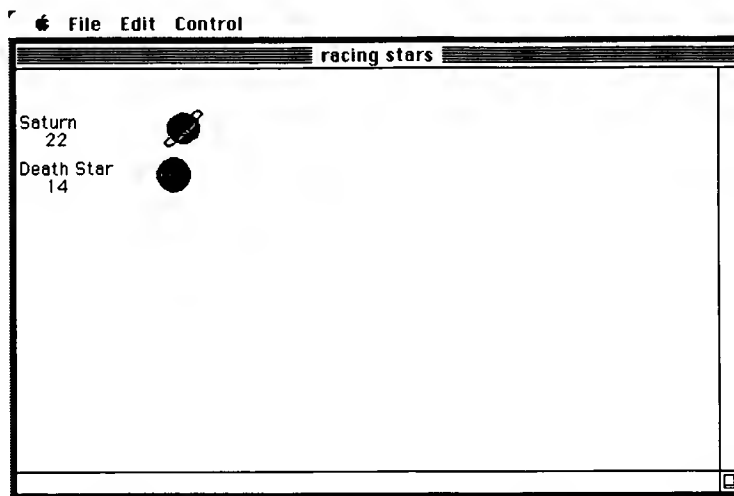


Illustration 8.30 Racing Stars program at beginning

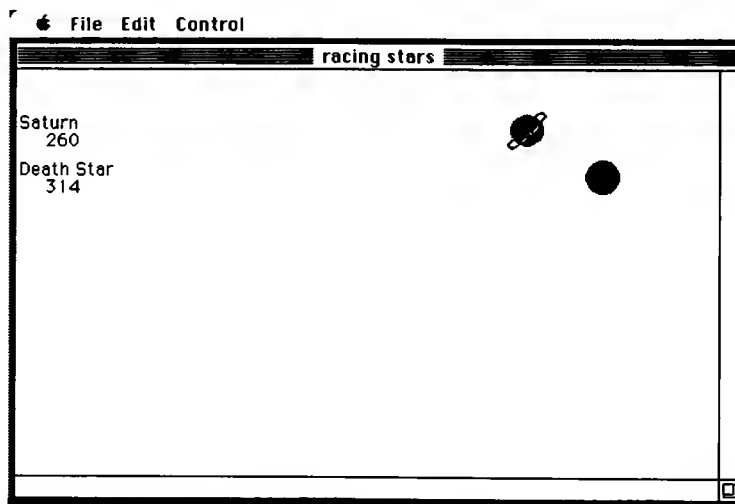


Illustration 8.31 Racing Stars, program at end

Listing, Racing Stars

```

10 ' racing stars
20 RANDOMIZE TIMER
30 DEFINT A-Z
40 DIM A(65),B(65)
50 CLS
60 A(0)=32:A(1)=32:B(0)=32:B(1)=32
70 A(2)=0:A(3)=0:B(2)=0:B(3)=0
80 A(4)=0:A(5)=0:B(4)=0:B(5)=0
90 A(6)=0:A(7)=0:B(6)=0:B(7)=0
100 A(8)=0:A(9)=&H70:B(8)=0:B(9)=0
110 A(10)=0:A(11)=&H1F8:B(10)=7:B(11)=&HE000
120 A(12)=7:A(13)=&HE30C:B(12)=&H1F:B(13)=&HFC00
130 A(14)=&H1F:A(15)=&HFE0C:B(14)=&H7F:B(15)=&HFE00
140 A(16)=&H7F:A(17)=&HFE18:B(16)=&HFF:B(17)=&HFF00
150 A(18)=&HFF:A(19)=&HFF30:B(18)=&H1FF:B(19)=
    &HFF80
160 A(20)=&H3FF:A(21)=&HFFE0:B(20)=&H3FF:B(21)=
    &HFFC0
170 A(22)=&H30F:A(23)=&HFFA0:B(22)=&H3FF:B(23)=
    &HFFC0
180 A(24)=&H7FF:A(25)=&HFF60:B(24)=&H7FF:B(25)=
    &HFFE0
190 A(26)=&H7FF:A(27)=&HFEE0:B(26)=&H76F:B(27)=
    &HFFE0
200 A(28)=&HFFF:A(29)=&HFEF0:B(28)=&HFFF:B(29)=
    &HFFF0
210 A(30)=&HFFF:A(31)=&HFCF0:B(30)=&HFFF:B(31)=
    &HFFF0

```

```

220 A(32)=&HFFF:A(33)=&HF9F0:B(32)=&HEFB:B(33)=
    &HFFF0
230 A(34)=&HFFF:A(35)=&HF3F0
240 A(36)=&HFFF:A(37)=&H9FF0
250 A(38)=&HFFF:A(39)=&HCFF0
260 A(40)=&H7FE:A(41)=&H3FE0
270 A(42)=&H7FF:A(43)=&HDFE0
280 A(44)=&H1FC:A(45)=&HFFC0
290 A(46)=&H7F9:A(47)=&HFFC0
300 A(48)=&HCF3:A(49)=&HFF00
310 A(50)=&H186F:A(51)=&HFC00
320 A(52)=&H303F:A(53)=&HF800
330 A(54)=&H30C7:A(55)=&HE000
340 A(56)=&H1F80:A(57)=0
350 A(58)=&HE00:A(59)=0
360 A(60)=0:A(61)=0
370 A(62)=0:A(63)=0
380 A(64)=0:A(65)=0
390 FOR I=34 TO 64 STEP 2
400 B(I)=B(66-I):B(I+1)=B(67-I)
410 NEXT I
420 S=1:P=S*32
430 CALL MOVETO(2,41):PRINT "Saturn"
440 CALL MOVETO(2,41+P):PRINT "Death Star"
450 X1=80:Y1=25:X2=X1+S*32-1:Y2=Y1+S*32-1
460 X3=X1:X4=X3+P-1:Y3=Y1+P:Y4=Y3+P-1
470 D=1:F=700/D
480 FOR I=1 TO F
490 PUT(X1,Y1)-(X2,Y2),A,PSET
500 PUT(X3,Y3)-(X4,Y4),B,PSET
510 IF RND(1)>.5 THEN X2=X2+D:X1=X1+D ELSE
    X4=X4+D:X3=X3+D
520 CALL MOVETO(12,53):PRINT X1-80
530 CALL MOVETO(12,53+P):PRINT X3-80
540 NEXT I
550 IF INKEY$="" THEN 550 ELSE 60

```

This program is a take-off on the previous one. Here we define two stars, the original ringed one, and the other a simple sphere with several white spots in it as highlights. The definition of this new object is found in lines 60 through 410 as hex constants placed into the array B, DIMensioned 65. The array A remains unchanged. Notice how the bottom half of the new star, B(34) through B(65) are defined as the upside-down definition of B(2) through B(33). You can do this with any symmetrical object.

After the two objects' definitions, the program prints an identifying name on the screen, and loops through PUTs of A and B across the screen. The trick is in line 510, where the horizontal displacement of each star is increased by D at random. This is what makes the race interesting. When the loop in lines 480-540 is through and the race is over, there is no way to predetermine which star will win.

Approaching Star program

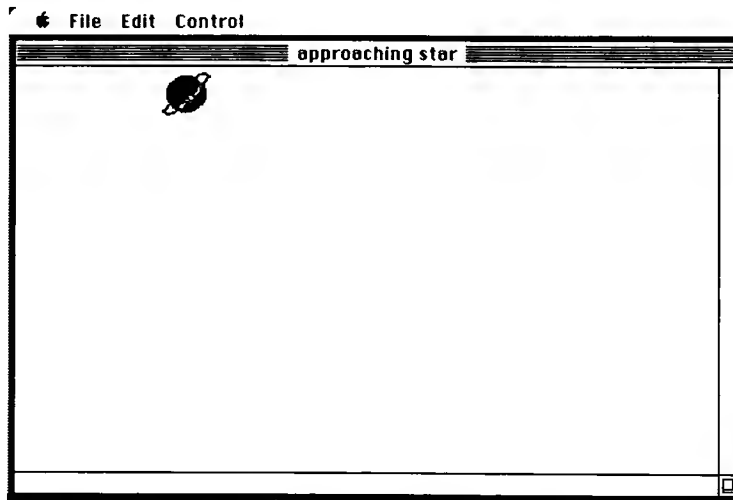


Illustration 8.32 Approaching Star from far away

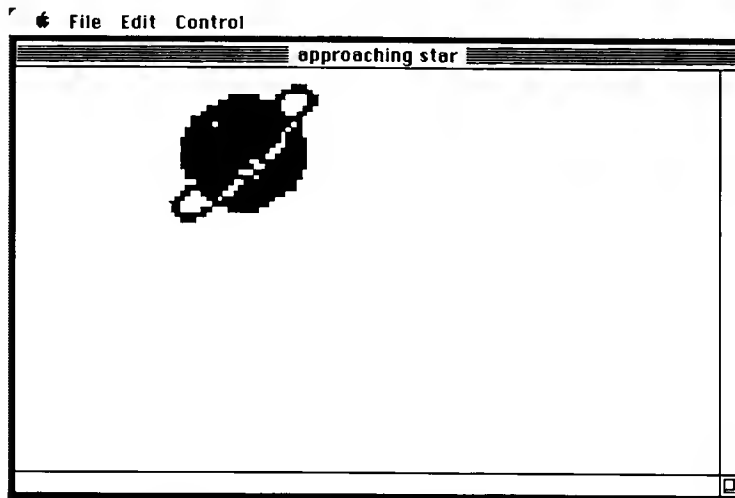


Illustration 8.33 Approaching Star closer

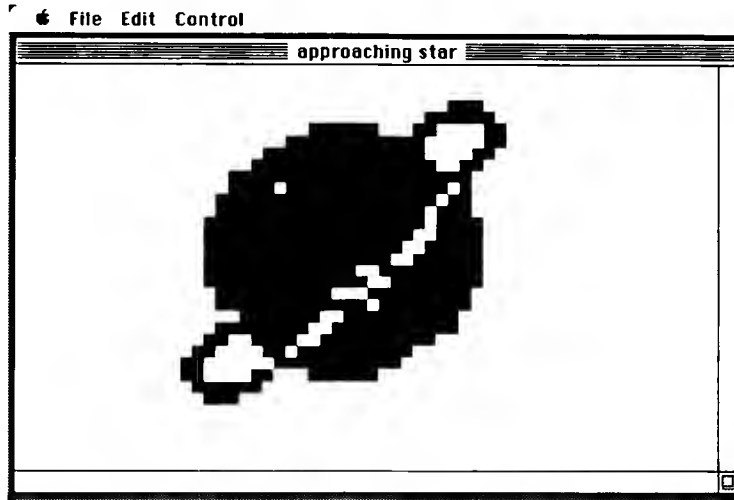


Illustration 8.34 Approaching Star very close

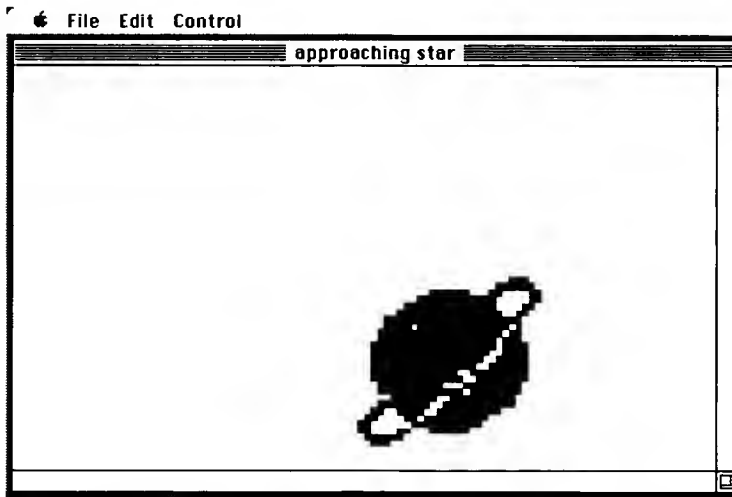


Illustration 8.35 Approaching Star receding

Listing, Approaching Star program

```
10 ' approaching star
20 DEFINT A-Z
30 DIM A(65),G(5)
40 CLS
50 A(0)=32:A(1)=32
60 A(2)=0:A(3)=0
70 A(4)=0:A(5)=0
80 A(6)=0:A(7)=0
90 A(8)=0:A(9)=&H70
100 A(10)=0:A(11)=&H1F8
110 A(12)=7:A(13)=&HE30C
120 A(14)=&H1F:A(15)=&HFE0C
130 A(16)=&H7F:A(17)=&HFE18
140 A(18)=&HFF:A(19)=&HFF30
150 A(20)=&H3FF:A(21)=&HFFE0
160 A(22)=&H3DF:A(23)=&HFFA0
170 A(24)=&H7FF:A(25)=&HFF60
180 A(26)=&H7FF:A(27)=&HFEE0
190 A(28)=&HFFF:A(29)=&HFEF0
200 A(30)=&HFFF:A(31)=&HFCF0
210 A(32)=&HFFF:A(33)=&HF9F0
220 A(34)=&HFFF:A(35)=&HF3F0
230 A(36)=&HFFF:A(37)=&H9FF0
240 A(38)=&HFFF:A(39)=&HCFF0
250 A(40)=&H7FE:A(41)=&H3FE0
260 A(42)=&H7FF:A(43)=&HDFE0
270 A(44)=&H1FC:A(45)=&HFFC0
280 A(46)=&H7F9:A(47)=&HFFC0
290 A(48)=&HCF3:A(49)=&HFF00
300 A(50)=&H186F:A(51)=&HFC00
310 A(52)=&H303F:A(53)=&HF800
320 A(54)=&H30C7:A(55)=&HE000
330 A(56)=&H1F80:A(57)=0
340 A(58)=&HE00:A(59)=0
350 A(60)=0:A(61)=0
360 A(62)=0:A(63)=0
370 A(64)=0:A(65)=0
380 S=1
390 X1=100:Y1=0:X2=X1+31:Y2=Y1+31
400 D=1:E=0
410 FOR I=0 TO 500
420 PUT(X1,Y1)-(X2,Y2),A,PSET
430 IF I=260 THEN D=0:E=1
440 IF I>245 AND I<255 THEN E=0:D=0
450 X2=X2+D:X1=X1+E
460 Y2=Y2+D:Y1=Y1+E
470 NEXT I
480 IF INKEY$="" THEN 480 ELSE 40
```

This program uses only one image definition, the same one as has existed right along, the ringed planet. The key difference is within the loop in lines 410-470. Here the PUT in line 420 will be executed 500 times, but the values for the rectangle's coordinates not only shift diagonally across the screen, but they redefine the size of the rectangle.

Look at line 400, where the upper left coordinate's displacement D starts as 1, and the lower right coordinate's displacement E starts as 0. Until I is greater than 245 (see line 440) the image grows and seems to come toward you from the upper left because D is fixed, while the lower right coordinate moves down and to the right because E is 1. When I is between 245 and 255, the image pauses because both displacements are set to zero. Later, when I is 260 or larger, the lower right coordinate is the one that is fixed, while the upper left one approaches it. This produces the effect of recession, because the star is shrinking toward the lower right.

Revolving Stars program

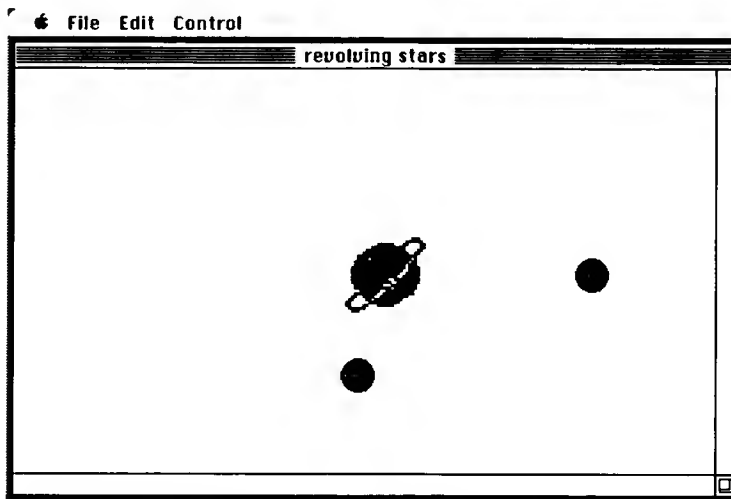


Illustration 8.36 Revolving stars, first view

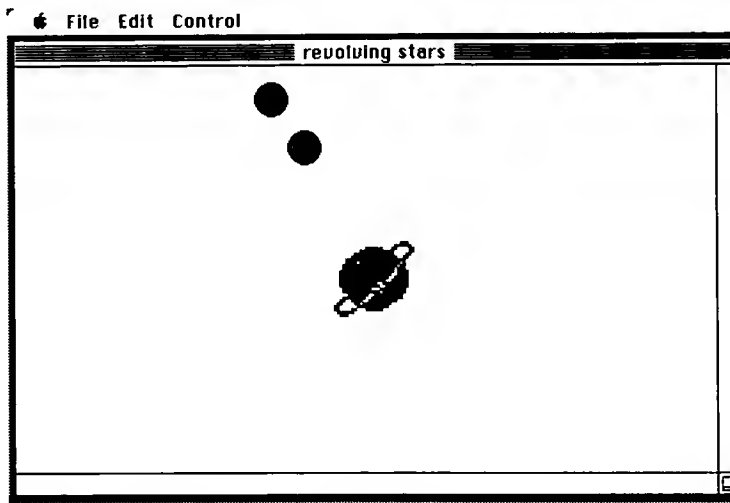


Illustration 8.37 Revolving stars, second view

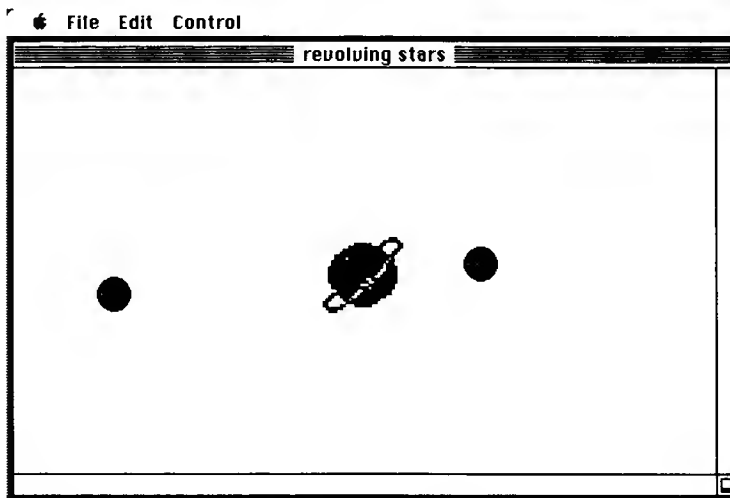


Illustration 8.38 Revolving stars, third view

Listing, Revolving Stars

```

10 ' revolving stars
20 DEFINT A-G
30 DIM A(65),B(65)
40 CLS
50 A(0)=32:A(1)=32
60 A(2)=&H7F:A(3)=&H0
70 A(4)=&H1FF:A(5)=&HC000

```

```

80 A(6)=&H7FF:A(7)=&HF000
90 A(8)=&HFFF:A(9)=&HF800
100 A(0)=32:A(1)=32:B(0)=32:B(1)=32
110 A(2)=0:A(3)=0:B(2)=0:B(3)=0
120 A(4)=0:A(5)=0:B(4)=0:B(5)=0
130 A(6)=0:A(7)=0:B(6)=0:B(7)=0
140 A(8)=0:A(9)=&H70:B(8)=0:B(9)=0
150 A(10)=0:A(11)=&H1F8:B(10)=7:B(11)=&HE000
160 A(12)=7:A(13)=&HE30C:B(12)=&H1F:B(13)=&HFC00
170 A(14)=&H1F:A(15)=&HFE0C:B(14)=&H7F:B(15)=&HFE00
180 A(16)=&H7F:A(17)=&HFE18:B(16)=&HFF:B(17)=&HFF00
190 A(18)=&HFF:A(19)=&HFF30:B(18)=&H1FF:B(19)=
    &HFF80
200 A(20)=&H3FF:A(21)=&HFFE0:B(20)=&H3FF:B(21)=
    &HFFC0
210 A(22)=&H3DF:A(23)=&HFFA0:B(22)=&H3FF:B(23)=
    &HFFC0
220 A(24)=&H7FF:A(25)=&HFF60:B(24)=&H7FF:B(25)=
    &HFFE0
230 A(26)=&H7FF:A(27)=&HFEE0:B(26)=&H76F:B(27)=
    &HFFE0
240 A(28)=&HFFF:A(29)=&HFEF0:B(28)=&HFFF:B(29)=
    &HFFF0
250 A(30)=&HFFF:A(31)=&HFCF0:B(30)=&HFFF:B(31)=
    &HFFF0
260 A(32)=&HFFF:A(33)=&HF9F0:B(32)=&HEFB:B(33)=
    &HFFF0
270 A(34)=&HFFF:A(35)=&HF3F0
280 A(36)=&HFFF:A(37)=&H9FF0
290 A(38)=&HFFF:A(39)=&HCF0
300 A(40)=&H7FE:A(41)=&H3FE0
310 A(42)=&H7FF:A(43)=&HDFE0
320 A(44)=&H1FC:A(45)=&HFFC0
330 A(46)=&H7F9:A(47)=&HFFC0
340 A(48)=&HCF3:A(49)=&HFF00
350 A(50)=&H186F:A(51)=&HFC00
360 A(52)=&H303F:A(53)=&HF800
370 A(54)=&H30C7:A(55)=&HE000
380 A(56)=&H1F80:A(57)=0
390 A(58)=&HE00:A(59)=0
400 A(60)=0:A(61)=0
410 A(62)=0:A(63)=0
420 A(64)=0:A(65)=0
430 FOR I=34 TO 64 STEP 2
440 B(I)=B(66-I):B(I+1)=B(67-I)
450 NEXT I
460 S1=1:S2=2:S3=1
470 XA=220:YA=120:XB=220:YB=105:XC=220:YC=115
480 PI=4*ATN(1):PIDEL1=PI/130:PIDEL2=PI/220:PIDEL3=
    PI/90

```

(continued)

```
490 T1=PI/13:T2=PI/7:T3=T2+PI/4
500 R1=120:R2=10:R3=90
510 X1=XA+R1*1.4*COS(T1):Y1=YA+R1*SIN(T1):X2=
    X1+S1*32-1:Y2=Y1+32*S1-1
520 X3=XB+R2*COS(T2):Y3=YB+R2*SIN(T2):X4=
    X3+S2*32-1:Y4=Y3+32*S2-1
530 X5=XC+R3*COS(T3):Y5=YC+R3*.9*SIN(T3):X6=
    X5+S3*32-1:Y6=Y5+32*S3-1
540 PUT(X1,Y1)-(X2,Y2),B,PSET
550 PUT(X3,Y3)-(X4,Y4),A,PSET
560 PUT(X5,Y5)-(X6,Y6),B,PSET
570 T1=T1+PIDEL1:T2=T2+PIDEL2:T3=T3+PIDEL3
580 GOTO 510
```

This program uses two planets, as did the racing stars. In this program, though, one of the planets is used in two different places on the screen, so you have three objects to follow in their motions. Instead of racing the three planets, we decided to make them revolve around each other. The largest, ringed, planet traces a small circle in the center of the screen. The two smaller ones revolve around it in opposite directions, one closer than the other.

The effect is indeed striking, and it is hard to describe either in words or pictures. If there is any one program in this book that you must run, it is this one.

The three planets are moved through "space" with the three PUT statements in lines 540-560. The ringed planet, you remember, is the image stored in array A. Array B holds the image for both other planets, as they are identical. We have set up the orbits of these three objects in lines 510-530, in which the X and Y displacements from the center of rotation are calculated. We will return to these calculations later. First, consider the overall problem as a list of goals:

- Each object must revolve in a fixed orbit.
- Object A (let's call it Alpha) will be defined within a rectangle with corner coordinates (X3,Y3)-(X4,Y4), it will revolve around a center at (XB,YB). Its orbit will be a circle, with radius R2.
- One object B (beta) will be in rectangle (X1,Y1)-(X2,Y2), it will revolve around a center (XA,YA), its orbit will be an ellipse with eccentricity 1.4, and the orbit's radius will be R1.
- The other object B (we'll call it gamma) will be in rectangle (X5,Y5)-(X6,Y6), with center (XC,YC), elliptical orbit with eccentricity 0.9, and orbit radius of R3.

We could calculate the orbital velocity based on eccentricity and radius. If we were especially ambitious, we could take into account the masses and positions of the neighboring objects. To keep it simple, we let the three objects take on the motions described above, with three orbital velocities chosen by programmer whim.

Now, let's identify the lines that do all this.

460	Define scales of the objects.
470	Define the center coordinates for all three objects.
480	Choose the incremental angles for rotation speed.
490	Establish the starting positions for each object.
500	Set the three centers of rotation.
510-580	Loop to revolve the planets.
510	Calculate the new corner coordinates for Alpha.
520	Calculate the new corner coordinates for Beta.
530	Calculate the new corner coordinates for Gamma.
540-560	Paint them in.
570	Increment the angles. Notice that Alpha's angle increment is negative, so it will revolve in the opposite direction from the other planets.
580	Repeat lines 510-570 <i>ad nauseam</i> .

We are continually amazed at the Macintosh's outstanding graphics capabilities. Even in BASIC, considering all of the statements that must be interpreted into machine code again and again, the machine operates at a sufficient speed to simulate motion. The sophistication of the BASIC itself is a great boon to this computer, because it takes advantage of so many of the Macintosh's superb Quickdraw routines.

CLOCKS

We have chosen clock faces as a theme for our graphics designs, because they allow a wide range of artistic freedom. It is not our intention to suggest that you convert your Macintosh into a clock. Rather, we recommend these application programs for your scrutiny as exercises in using some of the computer's more sophisticated graphics.

BASIC, through the use of Quickdraw CALLs, allows you to alter the output fonts in all of these ways. Illustration 9.1, 9.2 and 9.3 will indicate how you can do this.

CALL TEXTFONT(n)

Other fonts besides those in Illustration 9.1 should be available to you on the BASIC disk. You can discover them by investigating the systems font files.

n	Name	Optimum sizes	Description
0	Chicago	12	System font used for windows and menus
1	New York	9-24	Default BASIC output font
3	Geneva	9-24	System font (icon titles)
4	Monaco	9, 12	Non-proportional font

Illustration 9.1 TEXTFONT calls

CALL TEXTFACE (n)

The textfaces in Illustration 9.2 can be added together to provide more stylish options. For example, Bold-Italic can be specified with CALL TEXTFACE(3); Underline-Outline would be provided with a CALL TEXTFACE(12) because $12 = 8 + 4$.

bit #	value(n)	Description
0	1	Bold
1	2	<i>Italic</i>
2	4	<u>Underline</u>
3	8	Outline
4	16	Shadow
5	32	Condensed (squeeze characters)
6	64	Expanded (stretch characters)

Illustration 9.2 TEXTFACE calls

CALL TEXTSIZE(n)

This option is perhaps the most direct and easy to use. Simply use the fontsize you want — 9, 10, 12, 14, ... (up to 72 on some fonts)— as the argument in the CALL.

CALL TEXTMODE(n)

n	Mode	Description
0	Copy	Default mode. Text replaces contents of screen.
1	OR	Text superimposes screen image. Used to keep area around text from "whiting out" image below.
2	XOR	Text inverts existing image. You can "see through" text.
3	BIC	<u>B</u> lack <u>I</u> s <u>C</u> hanged. Text pixels are changed to white. The effect resembles the opposite of the OR mode.

Illustration 9.3 TEXTMODE calls

Application 1: Wall Clock

This program serves as a model for most programs in this chapter. Since it contains many of the features needed by most of the clock programs, we will describe it in more detail than the others.

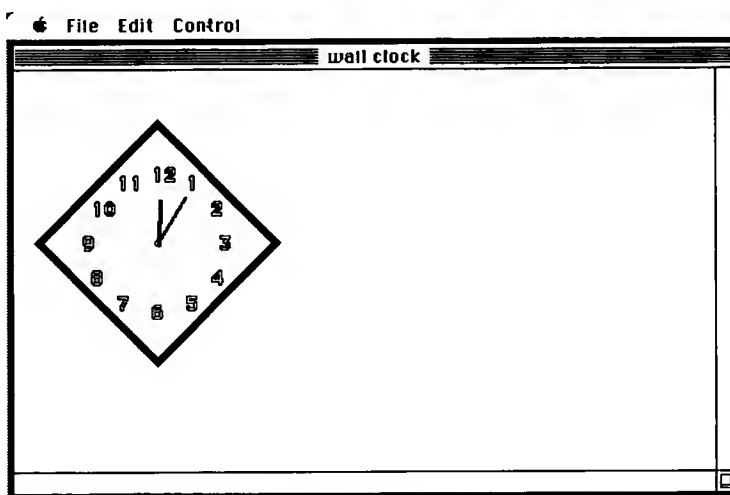


Illustration 9.4 Wall Clock program output

Listing, Wall Clock

```
10 ' wall clock driver
20 CLS
30 X=100:Y=120:F=0
40 GOSUB 1000
50 T$=MID$(TIME$,5,1)
60 IF INKEY$="/" THEN CALL MOVETO(5,300):STOP
70 IF T$=MID$(TIME$,5,1) THEN 60
80 GOSUB 1000:GOTO 50
90 GOTO 70
1000 ' wall clock
1010 P8=8*ATN(1)
1020 F0=1:FA=8:SI=12:MO=1:GOSUB 2000
1030 FOR J8=80 TO 86 STEP 2
1040 LINE(X,Y-J8)-(X-J8,Y):LINE(X-J8,Y)-(X,Y+J8)
1050 LINE(X,Y+J8)-(X+J8,Y):LINE(X+J8,Y)-(X,Y-J8)
1060 NEXT J8
1070 J8=3
1080 FOR I8=0 TO P8 STEP P8/12
1090 X8=48*COS(I8):Y8=48*SIN(I8)
1100 CALL MOVETO(X+X8-13,Y+Y8+5):PRINT J8
1110 J8=J8+1
1120 IF J8>12 THEN J8=1
1130 NEXT I8
1140 CIRCLE(X,Y),2,33,0,P8
1150 T8$=TIME$
1160 H8=VAL(LEFT$(T8$,2))
1170 M8=VAL(MID$(T8$,4,2))
1180 CALL PENSIZE(3,3)
1190 IF F=1 THEN LINE(X,Y)-(X+X7,Y+Y7),30
1200 IF F=1 THEN LINE(X,Y)-(X+X9,Y+Y9),30
1210 K8=(H8-3)*P8/12+M8*P8/720
1220 X7=30*COS(K8):Y7=30*SIN(K8)
1230 LINE(X,Y)-(X+X7,Y+Y7),33
1240 CALL PENSIZE(2,2)
1250 L8=(M8-15)*P8/60
1260 X9=37*COS(L8):Y9=37*SIN(L8)
1270 LINE(X,Y)-(X+X9,Y+Y9),33
1280 CALL PENNORMAL:F=1
1290 F0=1:FA=0:SI=12:MO=0:GOSUB 2000
1300 RETURN
2000 ' set text values
2010 CALL TEXTFONT(F0):CALL TEXTFACE(FA)
2020 CALL TEXTSIZE(SI):CALL TEXTMODE(MO)
2030 RETURN
```

Below is the outline of the program in pseudocode.

```

20      Clear screen
30      Define X and Y coordinates for clock's center. Set F, first CALL
        indicator, to 0.
40      Perform Wall Clock Routine. This first time around, the clock is
        drawn and the time displayed. From then on, because F will not
        be zero, the frame and digits of the clock are not drawn.
50      Isolate in T$ the second digit in the Minutes portion of TIME$. The
        system variable TIME$ has the format "HH:MM:SS" so the fifth
        character is the digit we want.
60      If user hits "/" then move cursor to bottom of screen, stop.
70      If the fifth character in TIME$ is still the same, check for user
        interrupt (go to line 60).
80      Time has advanced to next minute. Perform Wall Clock Routine,
        and go back to reset T$ in line 50.
1000     Wall Clock Routine
1010     Define P8 as  $2 * \pi$ . The angle whose tangent is 1 is  $45^\circ$ , which is
         $\pi/4$  radians. Therefore  $2 * \pi$  is  $8 * \text{ATN}(1)$ 
1020     Set FO (Font) to 1 (New York); FA (Face) to 8 (outlined); SI (size) to
        12; and MO (Mode) to 1 (Superimposed, ORed text). Then Per-
        form Set Text Values routine at line 2000.
1030-1060 Draw clock border or frame.
1070-1130 Print the numbers on the clock face. Notice that line 1100
        includes a slight shift to attempt to account for different number
        widths, because 10, 11, and 12 are twice as wide as the other
        numbers.
1140     Draw a small circle at center of clock. The hands will revolve
        around this center.
1150-1170 Place in H8 the hour and in M8 the minutes.
1180     Redefine size of pen for drawing hour hand to a 3x3 pixel point.
1190-1200 If  $F < > 0$  (not first call) then erase old hands. This is done by
        drawing old hands in white (color = 30).
1210-1230 Calculate position of hour hand and draw it. K8 is angle of hour
        hand.  $M8 * P8 / 720$  is fraction of hour, (needed so that if time is
        1:30, hour hand is halfway between 1 and 2).
1240     Calculate L8, angle of minute hand.
1250     Redefine size of pen to draw a thinner minute hand, 2x2 pixels.
1260     Calculate coordinates for tip of minute hand.
1270     Draw minute hand in black (color = 33).
1280     Reset pen size to normal, set F to 1 indicating hands drawn.

```

```

1290      Reset all TEXT values to system defaults: FO (Font) = 1 (New
          York); FA (Face) = 0 (plain); SI (Size) = 12; and MO (Mode) = 0
          (copy mode). Then perform Set Text Values routine at line 2000.
1300      Return.
2000      Set Text Values routine.
2010      Using CALLs to Macintosh ROM, set font, face, size, and mode.
2020      Return.

```

In this program, the LINE command and the CIRCLE command were used. LINE is a coordinate-to-coordinate instruction. Therefore, when a line of length R has to be drawn from point X, Y at an angle A , some transformations must be calculated. In the program, lines 1220 and 1260 show these calculations. Here's the procedure for line 1220.

1. Line 1210 calculates $K8$, the angle of the hour hand, based on the values of $H8$ and $M8$, $H8$ and $M8$ are the hour and minute values derived from the system variable $TIME\$$. The angle is one twelfth of a full circle ($P8$) for each hour, plus one 720th of a full circle for each minute. Note that the hour corresponding to an angle of 0 is three o'clock, not noon, so we subtract three in that calculation.
2. Line 1220 calculates $X7$ and $Y7$, the horizontal and vertical displacements from the center of the clock. We calculate the endpoints Xy and $Y7$ using polar coordinates to draw a line of angle $K8$ and radius length 30.

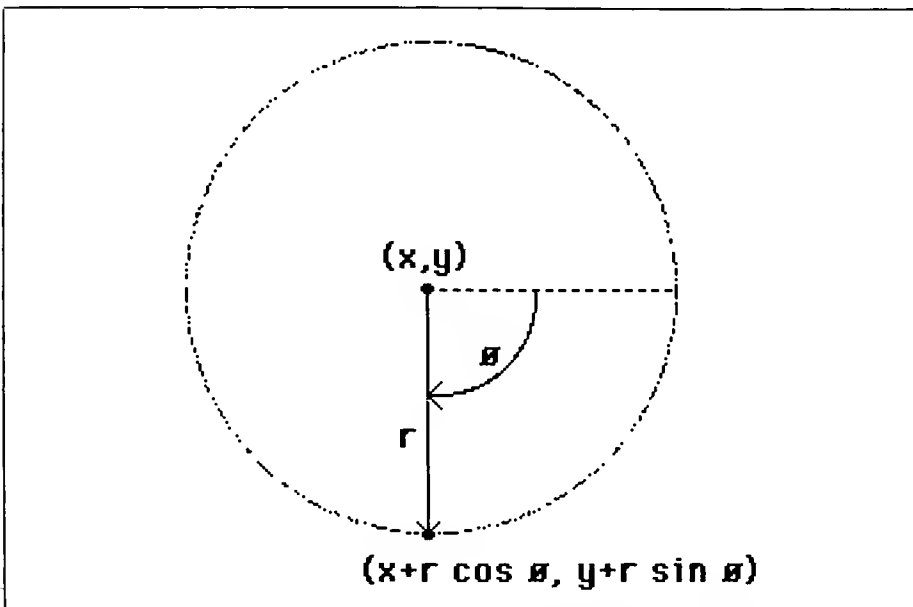


Illustration 9.5 X-distance and Y-distance calculations using the length of the hour hand as 30 and the angle as $K8$

3. Line 1230 draws the hour hand from the center of the clock X,Y to its proper position around the face, which is X,Y displaced by X7,Y7.

Other noteworthy aspects of this first clock program are the use of PENNORMAL and PENSIZE to change the thickness of lines drawn. We have found a reference which we recommend highly. The magazine *Macworld*, published by PC World Communications, Inc., at 555 De Haro St., San Francisco, CA 94107, is an invaluable aid for hints and tricks on the Macintosh. The July/August 1984 issue in particular has helped us, especially the article, *Open Window — An exchange of Macintosh discoveries*, by Tandy Trower, Microsoft's marketing manager for language products. In this article, the Toolbox Calls to the Macintosh ROM are discussed in enough detail (far more than in the BASIC manual) to use with confidence in your own BASIC programs. We urge you to subscribe to this magazine, and if you can't do that, at least get a copy of this particular issue.

Application 2: Digital Clock

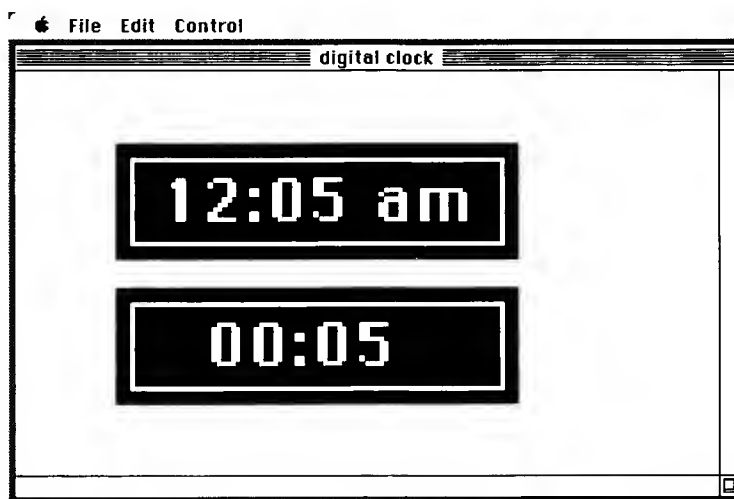


Illustration 9.6 The Macintosh digital clock

Listing, Digital Clock

```

10 ' digital clock driver
20 CLS
30 X=70
40 Y=50:M=0:GOSUB 1000
50 Y=150:M=1:GOSUB 1000
60 T$=MID$(TIME$,5,1)
70 IF INKEY$="/" THEN CALL MOVETO(5,300):STOP
80 IF T$=MID$(TIME$,5,1) THEN 70

```

(continued)


```
90 Y=50:M=0:GOSUB 1000
100 Y=150:M=1:GOSUB 1000
110 GOTO 60
1000 ' digital clock
1010 F0=0:FA=8:SI=40:M0=1:GOSUB 2000
1020 CALL PENSIZE(3,3)
1030 CALL MOVETO(X,Y)
1040 LINE(X,Y)-(X+280,Y+80),33,BF
1050 LINE(X+10,Y+10)-(X+270,Y+70),30,B
1060 CALL MOVETO(X+32,Y+54)
1070 T8$=TIME$
1080 H8=VAL(LEFT$(T8$,2))
1090 IF M=1 THEN T8$=" "+LEFT$(T8$,5):GOTO 1140
1100 IF H8<12 THEN S8$="am" ELSE S8$="pm"
1110 H8=H8 MOD 12
1120 IF H8=0 THEN H8=12
1130 T8$=RIGHT$(STR$(100+H8),2)+MID$(T8$,3,3)+"
    "+S8$
1140 PRINT T8$
1150 CALL PENNORMAL
1160 F0=1:FA=0:SI=12:M0=0:GOSUB 2000
1170 RETURN
2000 ' set text values
2010 CALL TEXTFONT(F0):CALL TEXTFACE(FA)
2020 CALL TEXTSIZE(SI):CALL TEXTMODE(M0)
2030 RETURN
```

The notes on the program below outline the steps involved in the program.

20	Clear screen
30-40	Define upper left corner of top display (12-hour clock) and set M=0, indicating 12-hour clock. Then perform Digital Clock Routine at line 1000.
50	Define upper left corner of bottom display (24-hour clock) and set M=1, indicating 24-hour clock. Then perform Digital Clock Routine at line 1000.
60	Get second digit in minutes portion of TIME\$.
70	If user hits "/" then move cursor out of way, stop.
80	If time hasn't changed, go back to line 70.
90	Perform Digital Clock routine for 12-hour (top) clock.
100	Perform Digital Clock routine for 24-hour clock.
110	Go to line 60 to check time.
1000	Digital Clock Routine.
1010	Set Font = Chicago, Face = Outline, Size = 40, Mode = Overlay (OR). Perform Set Text Values routine.
1020-1050	Set fat pen size, draw large black rectangle. Then move in 10 units, draw white rectangle.
1060	Position cursor for time to be displayed.

1070-1080	Get hours in H8.
1090	If 24-hour clock, simply print out first five characters of TIME\$ "HH:MM", so skip lines 1100-1130.
1100-1130	Set "am" or "pm" as appropriate, get 12-hour time, place leading zero if time is less than 10 o'clock.
1140	Print time. Note that the textface is outline and the textsize is 40. Whatever gets printed produces its own large black rectangle as a surrounding area.
1150-1160	Reset pen and text attributes to normal so that system messages and future program runs will be standard. Perform Set Text Values routine to do this.
1170	Return
2000	Set Text Values Routine.
2010-2020	Use Toolbox calls to set fonts
2020	Return.

Application 3: Two Clocks

This application is simply a combination of the two previous programs. The point of this exercise is to demonstrate the flexibility of these subroutines.

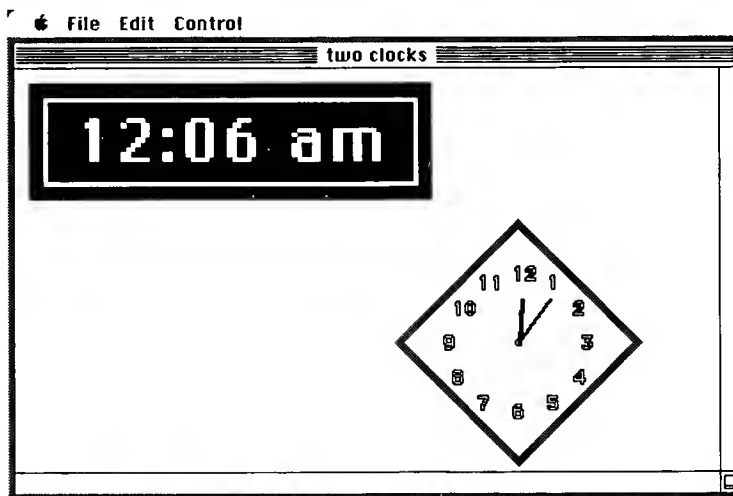


Illustration 9.7 Two Macintosh clocks

Listing, Two Clocks

```

10 ' two clocks driver
20 CLS
30 X=10:Y=10:M=0:GOSUB 2000
40 X=350:Y=190:GOSUB 1000
50 T$=MID$(TIME$,5,1)
60 IF INKEY$="/" THEN CALL MOVETO(5,300):STOP
70 IF T$=MID$(TIME$,5,1) THEN 60
80 X=10:Y=10:M=0:GOSUB 2000
90 X=350:Y=190:GOSUB 1000
100 GOTO 50
1000 ' wall clock
1010 P8=8*ATN(1)
1020 FO=1:FA=8:SI=12:MO=1:GOSUB 3000
1030 FOR J8=80 TO 86 STEP 2
1040 LINE(X,Y-J8)-(X-J8,Y):LINE(X-J8,Y)-(X,Y+J8)
1050 LINE(X,Y+J8)-(X+J8,Y):LINE(X+J8,Y)-(X,Y-J8)
1060 NEXT J8
1070 J8=3
1080 FOR I8=0 TO P8 STEP P8/12
1090 X8=48*COS(I8):Y8=48*SIN(I8)
1100 CALL MOVETO(X+X8-13,Y+Y8+5):PRINT J8
1110 J8=J8+1
1120 IF J8>12 THEN J8=1
1130 NEXT I8
1140 CIRCLE(X,Y),2,33,0,P8
1150 T8$=TIME$
1160 H8=VAL(LEFT$(T8$,2))
1170 M8=VAL(MID$(T8$,4,2))
1180 CALL PENSIZE(3,3)
1190 IF F=1 THEN LINE(X,Y)-(X+X7,Y+Y7),30
1200 IF F=1 THEN LINE(X,Y)-(X+X9,Y+Y9),30
1210 K8=(H8-3)*P8/12+M8*P8/720
1220 X7=30*COS(K8):Y7=30*SIN(K8)
1230 LINE(X,Y)-(X+X7,Y+Y7),33
1240 CALL PENSIZE(2,2)
1250 L8=(M8-15)*P8/60
1260 X9=37*COS(L8):Y9=37*SIN(L8)
1270 LINE(X,Y)-(X+X9,Y+Y9),33
1280 CALL PENNORMAL:F=1
1290 FO=1:FA=0:SI=12:MO=0:GOSUB 3000
1300 RETURN
2000 ' digital clock
2010 FO=0:FA=8:SI=40:MO=1:GOSUB 3000
2020 CALL PENSIZE(3,3)
2030 CALL MOVETO(X,Y)
2040 LINE(X,Y)-(X+280,Y+80),33,BF
2050 LINE(X+10,Y+10)-(X+270,Y+70),30,B
2060 CALL MOVETO(X+32,Y+54)

```

```

2070 T8$=TIME$
2080 H8=VAL(LEFT$(T8$,2))
2090 IF M=1 THEN T8$=" "+LEFT$(T8$,5):GOTO 2140
2100 IF H8<12 THEN S8$="am" ELSE S8$="pm"
2110 H8=H8 MOD 12
2120 IF H8=0 THEN H8=12
2130 T8$=RIGHT$(STR$(100+H8),2)+MID$(T8$,3,3)+"
      "+S8$
2140 PRINT T8$
2150 CALL PENNORMAL
2160 FO=1:FA=0:SI=12:MO=0:GOSUB 3000
2170 RETURN
3000 ' set text values
3010 CALL TEXTFONT(FO):CALL TEXTFACE(FA)
3020 CALL TEXTSIZE(SI):CALL TEXTMODE(MO)
3030 RETURN

```

In this program, lines 1000-1300 are unchanged from the Wall Clock you saw previously. Lines 2000-2170 are identical to lines 1000-1170 in the Digital Clock program. Even the checks for 12-hour or 24-hour clocks were left in. The Set Text Values routine was jogged to lines 3000-3030, again unchanged.

Application 4: Mantel Clock

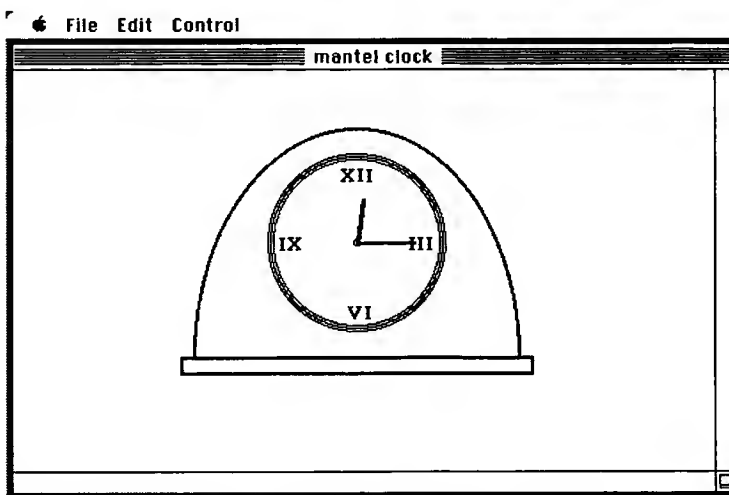


Illustration 9.8 Output from Mantel Clock program

Listing, Mantel Clock

```

10 ' mantel clock driver
20 CLS
30 X=240:Y=120:F=0
40 GOSUB 1000
50 T$=MID$(TIME$,5,1)
60 IF INKEY$="/" THEN CALL MOVETO(5,300):STOP
70 IF T$=MID$(TIME$,5,1) THEN GOTO 60
80 GOSUB 1000
90 GOTO 50
1000 ' mantel clock
1010 F0=9:FA=1:SI=12:M0=1:GOSUB 2000
1020 P8=8*ATN(1)
1030 FOR J8=58 TO 62 STEP 2
1040 CIRCLE(X,Y),J8,33,0,P8
1050 NEXT J8
1060 CIRCLE(X,Y),2,33,0,P8
1070 T8$=TIME$
1080 H8=VAL(LEFT$(T8$,2))
1090 M8=VAL(MID$(T8$,4,2))
1100 CALL PENSIZE(3,3)
1110 IF F=1 THEN LINE(X,Y)-(X+X7,Y+Y7),30
1120 IF F=1 THEN LINE(X,Y)-(X+X9,Y+Y9),30
1130 K8=(H8-3)*P8/12+M8*P8/720
1140 X7=30*COS(K8):Y7=30*SIN(K8)
1150 LINE(X,Y)-(X+X7,Y+Y7),33
1160 CALL PENSIZE(2,2)
1170 L8=(M8-15)*P8/60
1180 X9=40*COS(L8):Y9=40*SIN(L8)
1190 LINE(X,Y)-(X+X9,Y+Y9),33
1200 CALL PENSIZE(2,2)
1210 CIRCLE(X,Y+80),160,33,P8/2,P8,1.4
1220 LINE(X-123,Y+80)-(X+123,Y+92),33,B
1230 CALL PENNORMAL
1240 J8=1
1250 FOR I8=0 TO P8 STEP P8/4
1260 X8=48*COS(I8):Y8=48*SIN(I8)
1270 CALL MOVETO(X+X8-12,Y+Y8+6):PRINT MID$("IIII VI
      IXXII",3*(J8-1)+1,3)
1280 J8=J8+1
1290 NEXT I8
1300 F=1
1310 F0=1:FA=0:SI=12:M0=0::GOSUB 2000
1320 RETURN
2000 ' set text values
2010 CALL TEXTFONT(F0):CALL TEXTFACE(FA)
2020 CALL TEXTSIZE(SI):CALL TEXTMODE(M0)
2030 RETURN

```

Some selected remarks about this program:

- 1030-1050 Draw three circles with center X,Y, radius J8 (58, 60, 62), color=33 (black), starting at angle 0 and ending at angle P8 (two pi). The last two arguments in line 1040, start and end angles, are optional if you are going to draw a full circle. But if you want to draw only part of a circle, they must be specified.
- 1060-1190 Much the same as Wall Clock.
- 1210-1220 Draw clock frame. Notice that the CIRCLE command uses starting and ending angles of P8/2 and P8, and that the aspect of the circle is 1.4. This is what gives the mantel clock its oval outline.
- 1250-1290 Draw Roman numerals on clock face by printing them as strings on the face.

Application 5: News Room Clock

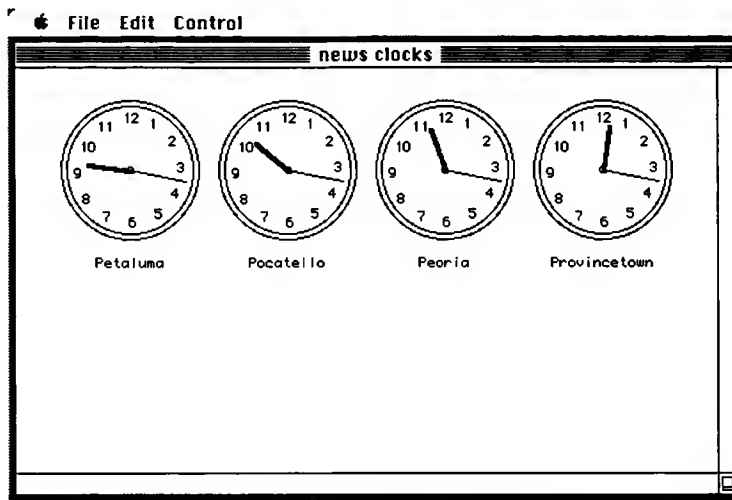


Illustration 9.9 Newsroom clocks a/a Macintosh

Listing, Newsroom Clocks

```

10 ' news room clock driver
20 CLS
30 DIM M$(12),N$(4)
40 FOR I=1 TO 12
50 READ M$(I)
60 NEXT I
70 DATA "Petaluma","Pomona","Pismo Beach"
80 DATA "Provo","Pueblo","Pocatello"
90 DATA "Pascagoula","Ponchatoula","Peoria"

```

(continued)

```
100 DATA "Provincetown","Portsmouth","Punxsutawney"
110 RAN00MIZE TIMER
120 X=-30:Y=70:F=0
130 FOR I=1 TO 4
140 N$(I)=M$(3*I-INT(3*RNO))
150 X=X+110
160 GOSUB 1000
170 NEXT I
180 T$=MIO$(TIME$,5,1)
190 IF INKEY$="/" THEN CALL MOVETO(5,300):STOP
200 IF T$=MIO$(TIME$,5,1) THEN 190
210 X=-30
220 FOR I=1 TO 4
230 X=X+110
240 GOSUB 1000
250 NEXT I
260 GOTO 180
1000 ' news room clocks
1010 P8=8*ATN(1)
1020 CALL TEXTMOOE(1)
1030 FOR J8=44 TO 48 STEP 4
1040 CIRCLE(X,Y),J8,33,0,P8
1050 NEXT J8
1060 CIRCLE(X,Y),2,33,0,P8
1070 T8$=TIME$
1080 H8=VAL(LEFT$(T8$,2))-4+1
1090 M8=VAL(MIO$(T8$,4,2))
1100 CALL PENSIZE(3,3)
1110 IF F>3 THEN LINE(X,Y)-(X+X7(I),Y+Y7(I)),30
1120 IF F>3 THEN LINE(X,Y)-(X+X9(I),Y+Y9(I)),30
1130 K8=(H8-3)*P8/12+M8*P8/720
1140 X7(I)=30*COS(K8):Y7(I)=30*SIN(K8)
1150 LINE(X,Y)-(X+X7(I),Y+Y7(I)),33
1160 L8=(M8-15)*P8/60
1170 X9(I)=40*COS(L8):Y9(I)=40*SIN(L8)
1180 CALL PENNORMAL
1190 LINE(X,Y)-(X+X9(I),Y+Y9(I)),33
1200 FO=4:FA=0:SI=9:MO=1:GOSUB 2000
1210 CALL TEXTFONT(4):CALL TEXTSIZE(9)
1220 J8=3
1230 FOR I8=-P8/100 TO P8-P8/100 STEP P8/12
1240 X8=36*COS(I8):Y8=36*SIN(I8)
1250 CALL MOVETO(X+X8-9,Y+Y8+4):PRINT J8
1260 J8=J8+1
1270 IF J8>12 THEN J8=1
1280 NEXT I8
1290 CALL TEXTMODE(0)
1300 IF F<4 THEN CALL MOVETO(X-3*LEN(N$(I)),Y+68):
      PRINT N$(I)
```

```

1310 F=F+1
1320 RETURN
2000 ' set text values
2010 CALL TEXTFONT(F0):CALL TEXTFACE(FA)
2020 CALL TEXTSIZE(S1):CALL TEXTMODE(M0)
2030 RETURN

```

The array M\$ holds twelve town names, three for each time zone. Lines 130-170 place into array N\$ four of these cities — one at random from the first three; another at random from the second three; a third at random from the third three; and a fourth at random from the fourth three in M\$. This gives variety to the display so that it is rare that you get two identical sets of four names in a given series of runs.

Lines 220-250 move each clock over from the preceding one.
 Lines 1230-1280 draw the numbers on the face of the clock whenever the hands have been drawn, because in lines 1110 and 1120 the previously drawn hands must be whited out, and that would leave a white streak in the numerals.

The displacement in the X and Y directions for the hour and minute hands (X7,Y7 for hour hand and X9,Y9 for minute hand) are DIMensioned to hold four different pairs of numbers (there are four clocks). Notice that lines 1110-1120 use these subscripted array values so that each clock's hour hand is different. In a newsroom display like this, where each clock represents the four different time zones, each clock's hour hand is displaced by one hour from the previous one. Only the hour hand's angle changes between clocks. Remember that the endpoints of both hands change between clock faces, so all endpoint coordinates must be kept.

Application 6: Egg Timer

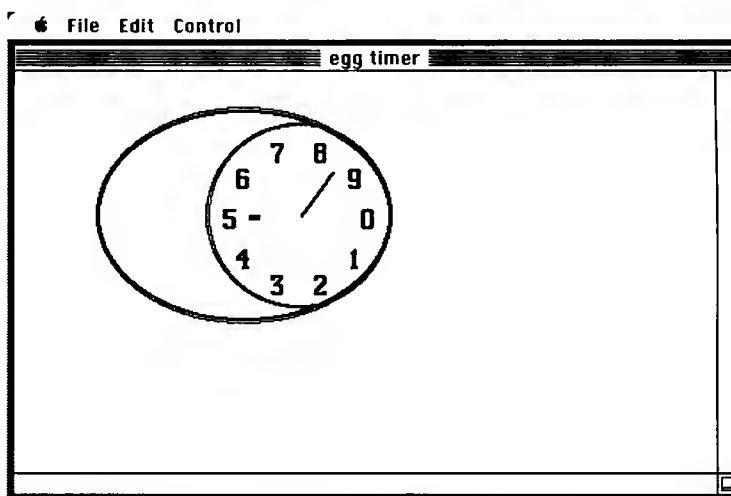


Illustration 9.10 Output from Egg Timer Program

Listing, Egg Timer

```
10 ' egg timer driver
20 CLS:F=0
30 INPUT "enter number of seconds . .":T
40 CLS
50 X=200:Y=100:V=T
60 WHILE V>0
70 IF INKEY$="/" THEN CALL MOVETO(5,300):STOP
80 GOSUB 1000
90 WEND
100 BEEP:BEEP:CALL MOVETO(10,250):STOP
1000 ' egg timer
1010 P8=8*ATN(1)
1020 IF F=1 THEN 1180
1030 F=1
1040 T8=TIMER
1050 F0=7:FA=1:SI=18:M0=1:GOSUB 2000
1060 FOR A8=1 TO 3
1070 CIRCLE(X-40,Y),100+A8,33,0,P8,.7+A8/100
1080 NEXT A8
1090 FOR A8=1 TO 2
1100 CIRCLE(X-A8,Y),62+A8,33,0,P8
1110 NEXT A8
1120 J8=0
1130 FOR I8=0 TO P8 STEP P8/10
1140 X8=48*COS(I8):Y8=48*SIN(I8)
1150 CALL MOVETO(X+X8-18,Y+Y8+10):PRINT J8
1160 J8=J8+1
1170 NEXT I8
1180 D8=TIMER-T8:V8=T-D8
1190 IF V8=V THEN RETURN ELSE V=V8
1200 M8=INT(V/60):S8=V-60*M8
1210 K8=M8*P8/10
1220 CALL PENSIZE(4,4)
1230 LINE(X+X6,Y+Y6)-(X+X7,Y+Y7),30
1240 X6=32*COS(K8):Y6=32*SIN(K8)
1250 X7=36*COS(K8):Y7=36*SIN(K8)
1260 LINE(X+X6,Y+Y6)-(X+X7,Y+Y7),33
1270 L8=S8*P8/60
1280 CALL PENSIZE(2,2)
1290 LINE(X,Y)-(X+X9,Y+Y9),30
1300 X9=37*COS(L8):Y9=37*SIN(L8)
1310 LINE(X,Y)-(X+X9,Y+Y9),33
1320 CALL PENNORMAL
1330 F0=1:FA=0:SI=12:M0=0:GOSUB 2000
1340 RETURN
2000 ' set text values
2010 CALL TEXTFONT(F0):CALL TEXTFACE(FA)
2020 CALL TEXTSIZE(SI):CALL TEXTMODE(M0)
2030 RETURN
```

Below are our notes on the Egg Timer Program.

20	Clear screen, set First Time Flag (F) to 0.
30	Get number of seconds T from user.
40	Clear screen of dialog.
50	Define X,Y as center of clock. Set timer V to T. V will go from T to 0, and as long as it is positive, the clock will run.
60-90	While there are still some seconds left in V, continue to perform the Egg Timer routine at line 1000.
100	When V reaches 0, beep the speaker, move the cursor out of the way, and stop.
1000	Egg timer routine.
1010	Define $2 * \pi$, a full circle in radians.
1020-1170	If F = 0 then set it to 1 and draw frame of egg timer.
1050	Set Font = Athens, Face = Bold, Size = 18, Mode = overlay (OR).
1060-1080	Draw egg shape ellipse using CIRCLE.
1090-1110	Draw circles around clock face.
1130-1170	Place digits 0 through 9 at six minute intervals.
1180	Set D8, seconds since last call to egg timer.
1190	V8 is seconds left. If V, return else set V = V8.
1200	M8 is number of full minutes left. S8 is number of seconds left in partial minutes.
1210-1260	Position minutes left blob on egg timer.
1270-1310	Erase and redraw the second hand.
1320-1340	Restore all font attributes and return.
2000	Set text values routine. No change from all others.

Conclusion

In this chapter you saw how the LINE and CIRCLE commands can help you to draw figures of many kinds. In particular, you saw how the CIRCLE command is perhaps a little misnamed, because it really draws ellipses. The circle is, after all, a special case of ellipse with eccentricity of 0 (what Microsoft's documentation on CIRCLE calls the *aspect* of the circle). The CIRCLE command is also flexible because you can draw a portion of a circle, or an arc. LINE and CIRCLE used together allowed us to draw that nicely shaped half-oval outline for the mantel clock.

In Chapter 12, you will see another way to form circle parts with the Toolbox calls *do* ARC and *do* OVAL, where *do* can be FRAME, PAINT, ERASE, INVERT, and FILL. We don't use all of the options, only enough to give you a taste of their power.

THE LINE COMMAND

The MAC's graphics instructions are good, and its resolution is marvellous. However, we miss the DRAW command that Microsoft provided for the TRS-80 Color Computer. Microsoft did provide DRAW in its BASIC version called GWBASIC (Gee Whiz BASIC), but the DRAW command is not on the MAC. Fear not, we will give you the DRAW command as a subroutine later in this chapter.

135

Using Angle and Radius with LINE

Suppose you want your line to start at a point (X1,Y1) and you want it to be R units long, as in a circle's radius, and to form the angle A with the horizontal line (Illustration 10.1).

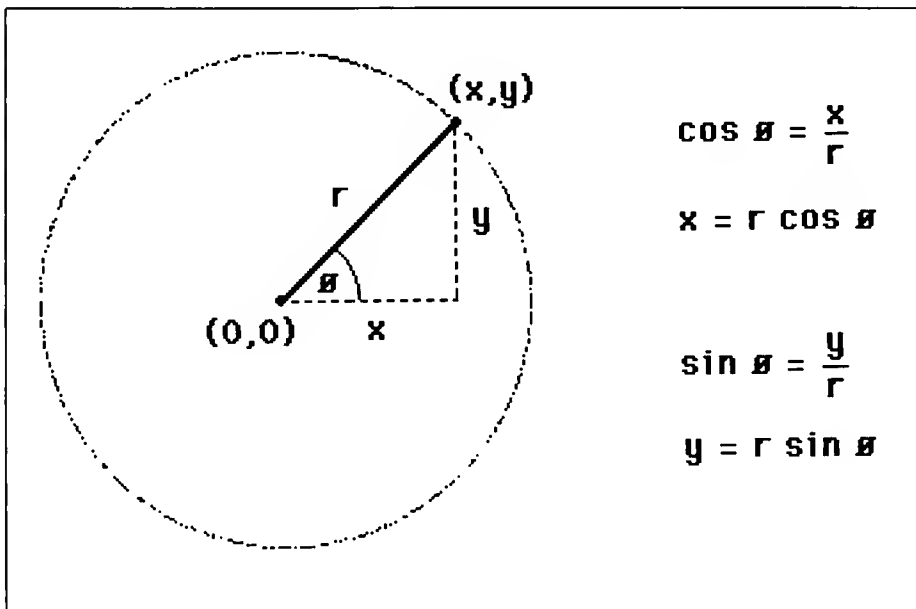


Illustration 10.1 Sketch of line in polar coordinates.

In this case, you don't know the X-Y displacement from the original point, so you can't use the straightforward

LINE (X1,Y1) – (X2,Y2)

because you don't know (X2,Y2). You can calculate these coordinates using elementary trigonometry. The horizontal distance from (X1,Y1) to the endpoint of the line is

$$DX = R * \cos(A)$$

where A is the angle in radians (Illustration 10.2).

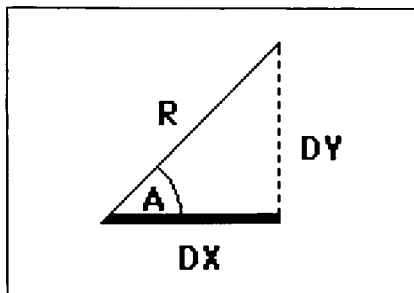


Illustration 10.2 Triangle with DX highlighted

The vertical distance from (X1,Y1) to the endpoint of the line is

$$DY = R \cdot \sin(A)$$

again with A in radians (Illustration 10.3).

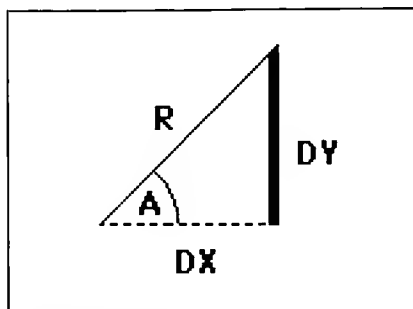


Illustration 10.3 Triangle with DY highlighted

Now, you can draw your line with the LINE command as

LINE (X1,Y1) – (X1 + DX,Y1 + DY)

If the cursor is already at (X1,Y1), it's even simpler.

LINE STEP (DX,DY)

The problem with both of these examples is that in either case you have to calculate the displacements DX and DY using the trigonometric sine and cosine functions. These are substantially slower to execute than a simple addition, or even several multiplications.

Advanced Applications of LINE

Calculating sines and cosines to determine the endpoint of a line is one of the several justifications for the DRAW command. With it, we can draw a line of a given length at any one of the eight angles (0, 45, 90, 135, 180, 225, 270, and 325 degrees) without using BASIC's SIN and COS functions. Before we do so, however, let us explore the LINE command in more detail.

Tessellation

Tessellation is the process of filling in an area with a geometric pattern. You can think of the hexagonal beehive pattern as one that can completely fill a surface with no gaps, regardless of the surface's area. Simple tessellation patterns, such as squares or triangles, don't offer much in visual appeal. More complex patterns, when chosen to fill an area, can be quite stunning.

We have found that tessellation *at random* is in many ways, different and more exciting than regular tessellation. For example, it is a simple task to write a program that produces squares over the entire surface of the screen. But if the squares are produced to fill the screen randomly, it becomes a visual "game" to see which area gets filled next. Consider each of the examples of tessellation that follow, and explore this intriguing technique.

ANGLEWALK: Random Tessellation with LINE

The Random Walk Problem is a classic environment in which to explore random tessellation. This problem involves a drunkard's aimless motions near a lightpost. Suppose the drunkard is leaning against the lamppost and steps away one pace. Then he steps off one more pace, but in any one of eight directions, including back to the post (Illustration 10.4). What path does that drunkard leave in his wanderings?

Let's code the problem on the Macintosh, using the LINE command as a tracing mechanism for the drunkard's motions.

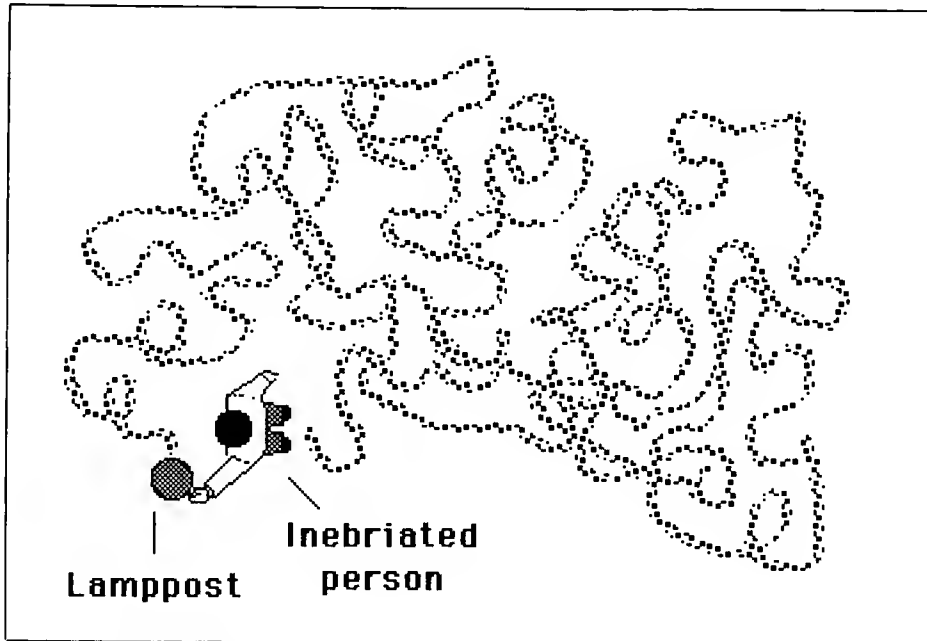


Illustration 10.4 Sketch of lamppost, drunkard, and random walk

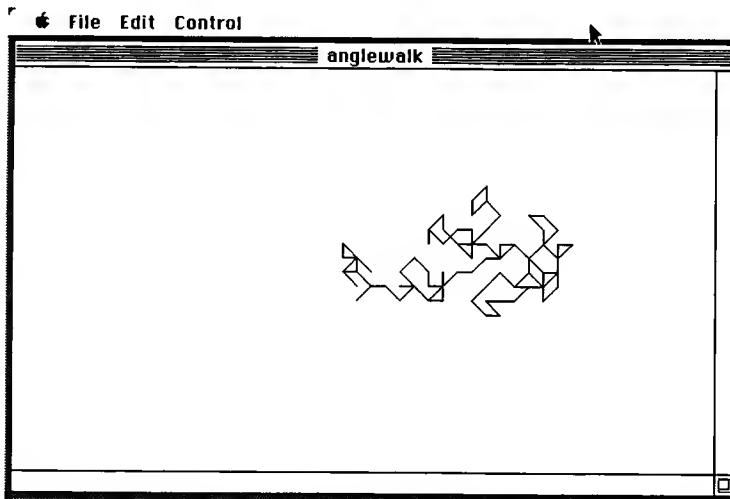


Illustration 10.5 Output of ANGLEWALK, $L = 10$

The output in Illustration 10.5 shows several key features of a random walk that is bounded on four sides. When the drunkard hits a wall, he doesn't learn to avoid that wall, and may hang around in the area for some time, as did ours in the example we show in Illustration 10.6.

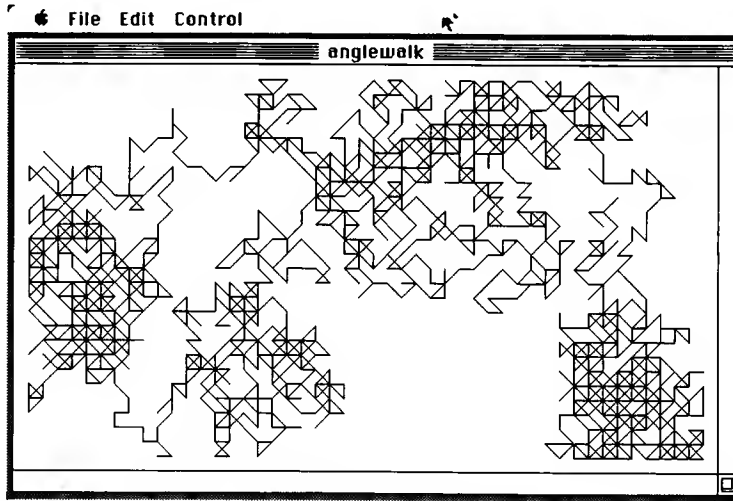


Illustration 10.6 Same program, much later

Notice that the lower right corner has been investigated thoroughly by our inebriant. His walk has left a pattern of squares with diagonals that resembles tiling. This is the beginnings of the tessellation of the complete screen.

Listing, ANGLEWALK

```

10 ' anglewalk
20 CLS: RANDOMIZE TIMER
30 LFB=10: RTB=480: UPB=10: LWB=270: L=10
40 X=250: Y=140: CALL MOVETO(X,Y): PSET(X,Y)
50 DX=L*(INT(RND*3)-1): DY=L*(INT(RND*3)-1): GOSUB
    1000
60 GOTO 50
1000 ' random line routine
1010 IF X+DX<LFB OR X+DX>RTB THEN DX=-DX
1020 IF Y+DY<UPB OR Y+DY>LWB THEN DY=-DY
1030 LINE -STEP(DX,DY)
1040 X=X+DX: Y=Y+DY
1050 RETURN

```

Below are notes on the ANGLEWALK program.

```

20      Clear screen and seed random number generator to whatever
        the internal clock contains.
30      Define the bounds of the screen. LFB, RTB, UPB, and LWB are
        the left, right, upper, and lower bounds of the screen respec-
        tively.
40      Move to the center of the screen and plant a point there.
50      Calculate random displacements DX and DY in the horizontal
        and vertical directions.  $RND*3$  produces a value between
        0.0000... and 2.999....  $INT(RND*3)$  therefore fetches a 0, 1, or 2.
        If we subtract 1 from that result, we end up with a - 1, 0, or + 1.
        Multiply that by L, the length of the pace, and we have defined
        both DX and DY as individually random displacements in both
        the X and Y directions. Now we call the subroutine to take the
        step.
60      This statement provides an infinite loop for the random walk.
1010-1020 Check if this added displacement DX or DY places the destina-
        tion outside the bounds of the screen. If so, simply reverse the
        direction of that displacement. Consider it the drunkard's reac-
        tion if he hits a wall, or if he is afraid of the dark beyond the
        immediate area around the lamppost. He'll either bounce or
        retreat a step.
1030      Draw that step.
1040      Reset X and Y, the place where the drunk is standing now.
1050      Return

```

Square Tessellation:

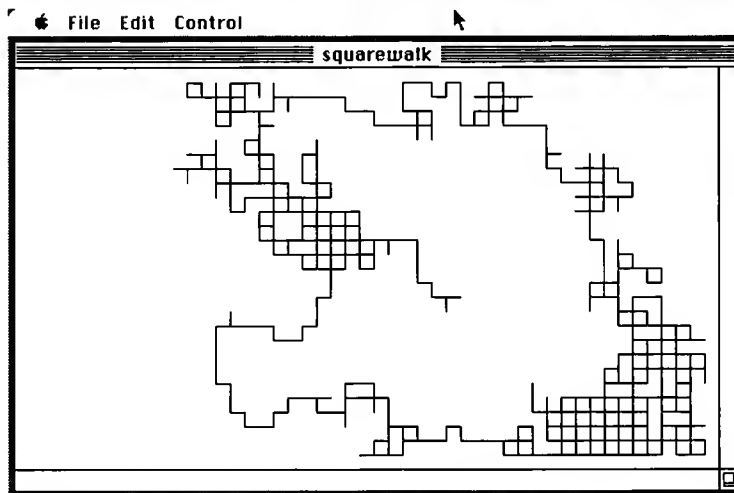


Illustration 10.7 Square tessellation

The only change we made to the ANGLEWALK program was to add this line in the subroutine:

```
1005 IF DX < > 0 AND DY < > 0 THEN RETURN
```

This forces either DX or DY or both to have a value of zero in order to draw the pace. This means that the pace will be either vertical or horizontal. You can modify it further by changing the pace length L from 10 to some other length.

Listing, SQUAREWALK

```
10 ' squarewalk
20 CLS: RANDOMIZE TIMER
30 LFB=10: RTB=480: UPB=10: LWB=270: L=10
40 X=250: Y=140: CALL MOVETO(X,Y): PSET(X,Y)
50 DX=L*(INT(RND*3)-1): DY=L*(INT(RND*3)-1): GOSUB
    1000
60 GOTO 50
1000 ' random line routine
1005 IF DX<>0 AND DY<>0 THEN RETURN ' modification
    to anglewalk
1010 IF X+DX<LFB OR X+DX>RTB THEN DX=-DX
1020 IF Y+DY<UPB OR Y+DY>LWB THEN DY=-DY
1030 LINE -STEP(DX,DY)
1040 X=X+DX: Y=Y+DY
1050 RETURN
```

Diamond Tessellation

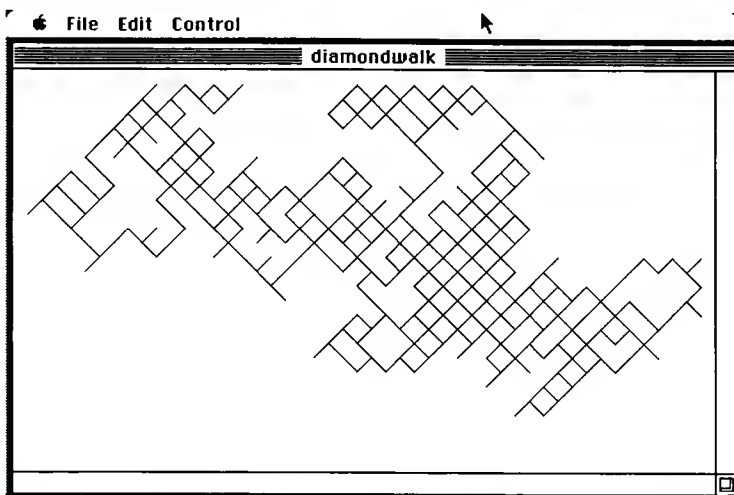


Illustration 10.8 Diamond tessellation

The step size was left at 10 for this pattern, and line 1005 was changed to

```
1005 IF DX=0 OR DY=0 THEN RETURN
```

which forces the step to stay on a diagonal.

Listing, DIAMONDWALK

```
10 ' diamondwalk
20 CLS: RANDOMIZE TIMER
30 LFB=10: RTB=480: UPB=10: LWB=270: L=10
40 X=250: Y=140: CALL MOVETO(X,Y): PSET(X,Y)
50 DX=L*(INT(RND*3)-1): DY=L*(INT(RND*3)-1): GOSUB
    1000
60 GOTO 50
1000 ' random line routine
1005 IF DX=0 OR DY=0 THEN RETURN ' modification to
    anglewalk
1010 IF X+DX<LFB OR X+DX>RTB THEN DX=-DX
1020 IF Y+DY<UPB OR Y+DY>LWB THEN DY=-DY
1030 LINE -STEP(DX,DY)
1040 X=X+DX: Y=Y+DY
1050 RETURN
```

Four-pointed Star Tessellation

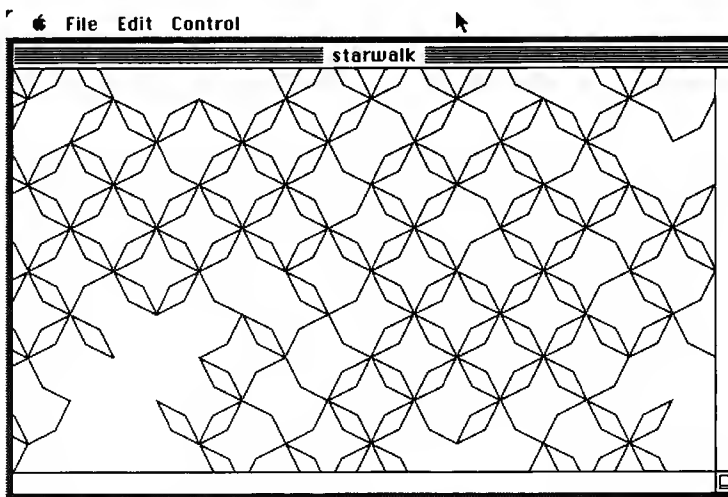


Illustration 10.9 Four-pointed stars output from STARWALK

Here we avoided both the 45-degree diagonals and the horizontal and vertical orientations. Rather, we forced the program to draw 30-degree and 60-degree lines. These angles cannot be drawn using the DRAW subroutine you will see later. We altered the line's angle with a new line 1030:

```
1030 LINE - STEP (2*DX,DY): LINE - STEP(DX,2*DY)
```

We kept line 1005 as before, to eliminate the horizontals and the verticals.

Listing, STARWALK

```
10 ' starwalk
20 CLS: RANDOMIZE TIMER
30 LFB=10: RTB=480: UPB=10: LWB=270: L=10
40 X=250: Y=140: CALL MOVETO(X,Y): PSET(X,Y)
50 OX=L*(INT(RND*3)-1): OY=L*(INT(RND*3)-1): GOSUB
    1000
60 GOTO 50
1000 ' random line routine
1005 IF DX=0 OR OY=0 THEN RETURN ' modification to
    anglewalk
1010 IF X+OX<LFB OR X+OX>RTB THEN OX=-OX
1020 IF Y+DY<UPB OR Y+DY>LWB THEN OY=-OY
1030 LINE -STEP(2*OX,OY): LINE -STEP(OX,2*OY) '
    modification
1040 X=X+OX+OX: Y=Y+OY+OY ' modification
1050 RETURN
```

Complex Tessellation

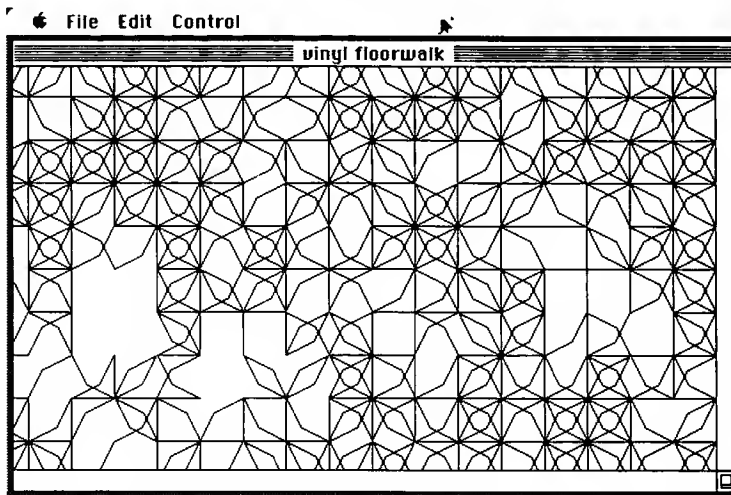


Illustration 10.10 Vinyl flooring tiles output, VINYL FLOORWALK

Illustration 10.10 shows the output if you remove line 1005 to allow horizontals and verticals back into the fray. Notice that the last two figures seem to go out of bounds. That's because the bounds check is on $X+DX$ and $Y+DY$ rather than $X+2*DX$ and $y+DY$. The Macintosh takes it in stride, never complaining about plotting points outside the bounds of the screen. However, don't try to remove lines 1010 and 1020 to simplify the process, because the plotting may get very busy outside of the screen's range, and that tends to slow down the action on the screen.

Listing, VINYL FLOORWALK

```

10 ' vinyl floorwalk
20 CLS: RANDOMIZE TIMER
30 LFB=10: RTB=480: UPB=10: LWB=270: L=10
40 X=250: Y=140: CALL MOVETO(X,Y): PSET(X,Y)
50 DX=L*(INT(RND*3)-1): DY=L*(INT(RND*3)-1): GOSUB
    1000
60 GOTO 50
1000 ' random line routine
1010 IF X+DX<LFB OR X+DX>RTB THEN DX=-DX
1020 IF Y+DY<UPB OR Y+DY>LWB THEN DY=-DY
1030 LINE -STEP(2*DX,DY): LINE -STEP(DX,2*DY)
1040 X=X+DX+DX: Y=Y+DY+DY
1050 RETURN

```

Suggestions:

- 1030 LINE -STEP(3*DX,DY): LINE -STEP(DX,3*DY)
- 1030 LINE -STEP(3*DX,2*DY): LINE -STEP(2*DX,3*DY)
- 1030 LINE -STEP(DX,DY): CIRCLE(X,Y),ABS(DX+DY)/3
- 1030 CIRCLE(X,Y),ABS(DX+DY)/3
- 25 DEFINT A-Z

We encourage you to explore this simple program further. You will discover a multitude of rewarding patterns, all different and all imbued with that captivating combination of randomness and symmetry.

Stars and Circles

We have modified the program in line 1030 three different ways to show you how so small a change can cause major changes in output. The first is STARS and CIRCLES.

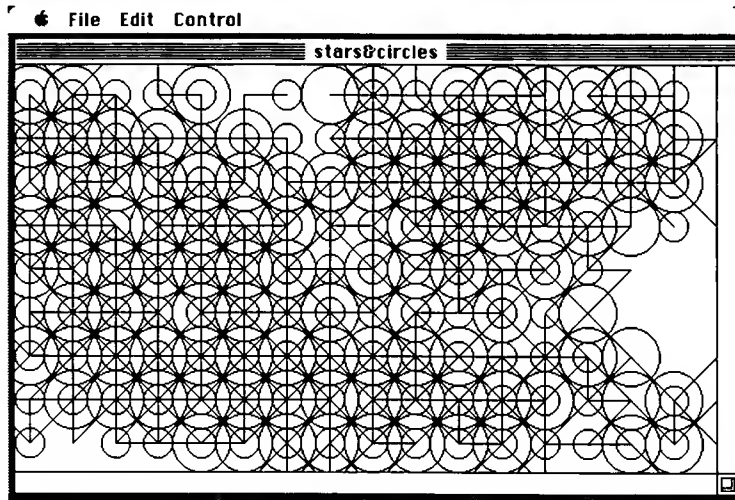


Illustration 10.11 Stars & Circles

Listing, STARS&CIRCLES

```

10 ' stars & circles
20 CLS: RANDOMIZE TIMER
30 LFB=10: RTB=480: UPB=10: LWB=270: L=30
40 X=250: Y=140: CALL MOVETO(X,Y): PSET(X,Y)
50 DX=L*(INT(RND*3)-1): DY=L*(INT(RND*3)-1): GOSUB
   1000
60 GOTO 50
1000 ' random circle routine
1010 IF X+DX<LFB OR X+DX>RTB THEN DX=-DX
1020 IF Y+DY<UPB OR Y+DY>LWB THEN DY=-DY
1030 LINE -STEP(DX,DY): CIRCLE(X,Y),ABS((DX+DY)/3)
1040 X=X+DX: Y=Y+DY
1050 RETURN

```

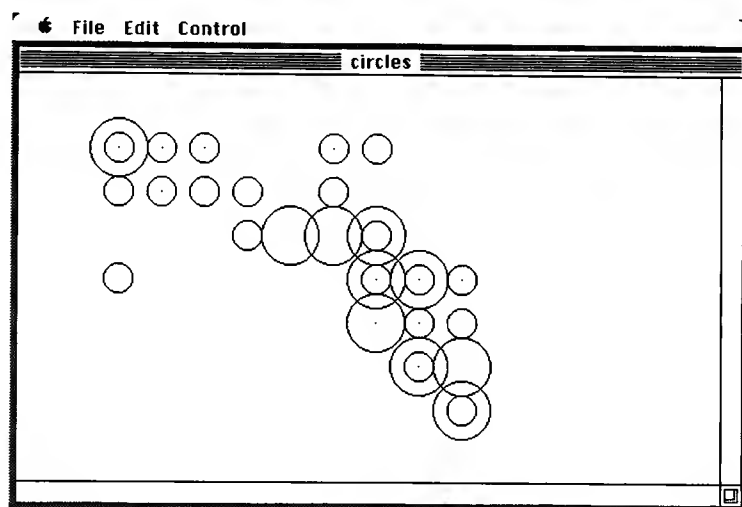


Illustration 10.12 Circles

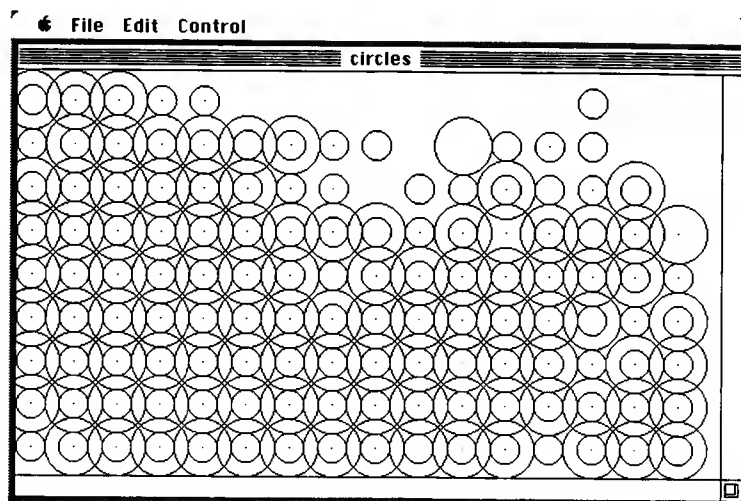


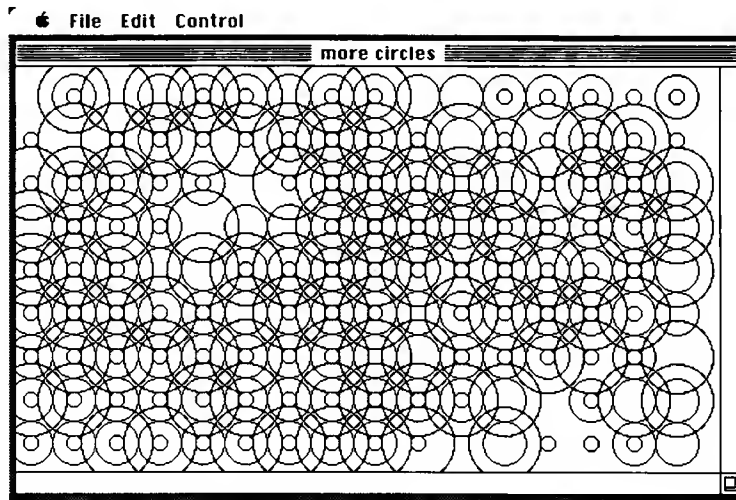
Illustration 10.13 Circles later

Listing, Circles

```

10 ' circles
20 CLS: RANDOMIZE TIMER
30 LFB=10: RTB=480: UPB=10: LWB=270: L=30
40 X=250: Y=140: CALL MOVETO(X,Y): PSET(X,Y)
50 DX=L*(INT(RND*3)-1): DY=L*(INT(RND*3)-1): GOSUB
    1000
60 GOTO 50
1000 ' random circle routine
1010 IF X+DX<LFB OR X+DX>RTB THEN DX=-DX
1020 IF Y+DY<UPB OR Y+DY>LWB THEN DY=-DY
1030 CIRCLE(X,Y),ABS((DX+DY)/3)
1040 X=X+DX: Y=Y+DY
1050 RETURN

```

**Illustration 10.14** More Circles**Listing, MORE CIRCLES**

```

10 ' more circles
20 CLS: RANDOMIZE TIMER
30 LFB=10: RTB=480: UPB=10: LWB=270: L=30
40 X=250: Y=140: CALL MOVETO(X,Y): PSET(X,Y)
50 DX=L*(INT(RND*3)-1): DY=L*(INT(RND*3)-1): GOSUB
    1000
60 GOTO 50
1000 ' random circle routine
1010 IF X+DX<LFB OR X+DX>RTB THEN DX=-DX
1020 IF Y+DY<UPB OR Y+DY>LWB THEN DY=-DY
1030 CIRCLE(X,Y),ABS((DX+DY)/3)+5
1040 X=X+DX: Y=Y+DY
1050 RETURN

```

Sierpinski Patterns

In the July 1984 issue of *Creative Computing*, (pages 148-180) was a most intriguing article by David Ahl on a class of patterns named after their originator, Sierpinski. We transferred the programs listed in the magazine relatively unchanged, and ran them on the Macintosh. We were rewarded with an incredible visual experience. What you see in this chapter is only the stale shadow of the program's real reward, which is to watch the pattern being generated on the screen. It's almost as if there were a live psychotic bug tracing these intriguing patterns within the computer.

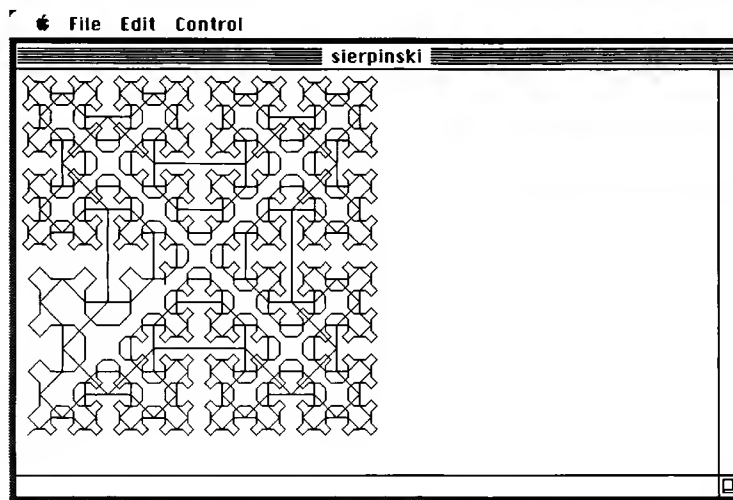


Illustration 10.15 Sierpinski

The program is a model of tight code, and it uses *recursion*, a feature of Microsoft BASIC that is not often found in other versions of this language. Recursion is the process of a routine calling itself until a condition is met, at which time it returns.

The listing of our version of this fine program is shown below. Note that line 230 in the subroutine starting at line 200 is

```
230 GOSUB 200: A = H: B = - H: GOSUB 800
```

This is an example of recursion in which a statement in the subroutine calls the subroutine itself. Line 210 is the escape from this seemingly infinite loop. If the condition is met, control is transferred back to the statement in line 230 following GOSUB 200, which is $A = H$. Control is returned to line 150 only when line 280 is executed.

Listing, Sierpinski

```
10 ' sierpinski
15 ' Creative Computing, July 1984 v10, #7, pp.
    148-160 David H. Ahl
20 DEFINT A-Z
30 CLS
40 FOR OI=1 TO 7
60 GOSUB 100
80 NEXT OI
90 GOTO 90
100 H0=512: SP=0: H=H0/4: X=H+H: Y=X+H: I=0
110 I=I+1: X=X-H: H=H/2: Y=Y+H
120 IF I<OI THEN 110
130 PS=I: GOSUB 600
140 GOSUB 200: A=H: B=-H: GOSUB 800
150 GOSUB 300: A=-H: B=-H: GOSUB 800
160 GOSUB 400: A=-H: B=H: GOSUB 800
170 GOSUB 500: A=H: B=H: GOSUB 800
180 GOSUB 700
190 RETURN
200 '
210 IF TP<=0 THEN RETURN
220 PS=TP-1: GOSUB 600
230 GOSUB 200: A=H: B=-H: GOSUB 800
240 GOSUB 300: A=H+H: B=0: GOSUB 800
250 GOSUB 500: A=H: B=H: GOSUB 800
260 GOSUB 200
270 GOSUB 700
280 RETURN
300 '
310 IF TP<=0 THEN RETURN
320 PS=TP-1: GOSUB 600
330 GOSUB 300: A=-H: B=-H: GOSUB 800
340 GOSUB 400: A=0: B=-2*H: GOSUB 800
350 GOSUB 200: A=H: B=-H: GOSUB 800
360 GOSUB 300
370 GOSUB 700
380 RETURN
400 '
410 IF TP<=0 THEN RETURN
420 PS=TP-1: GOSUB 600
430 GOSUB 400: A=-H: B=H: GOSUB 800
440 GOSUB 500: A=-2*H: B=0: GOSUB 800
450 GOSUB 300: A=-H: B=-H: GOSUB 800
460 GOSUB 400
```

```

470 GOSUB 700
480 RETURN
500 '
510 IF TP<=0 THEN RETURN
520 PS=TP-1: GOSUB 600
530 GOSUB 500: A=H: B=H: GOSUB 800
540 GOSUB 200: A=0: B=H+H: GOSUB 800
550 GOSUB 400: A=-H: B=H: GOSUB 800
560 GOSUB 500
570 GOSUB 700
580 RETURN
600 '
610 SP=SP+1: ST(SP)=PS
620 TP=PS: RETURN
700 '
710 SP=SP-1: TP=ST(SP): RETURN
800 '
810 LINE(X,Y)-(X+A,Y+B)
820 X=X+A: Y=Y+B: RETURN

```

The geometric design shown in Illustration 10.16 has square “points” at each corner. The next iteration produces a smaller design in the center of the first one; this smaller design has corners that are made up of images of itself.

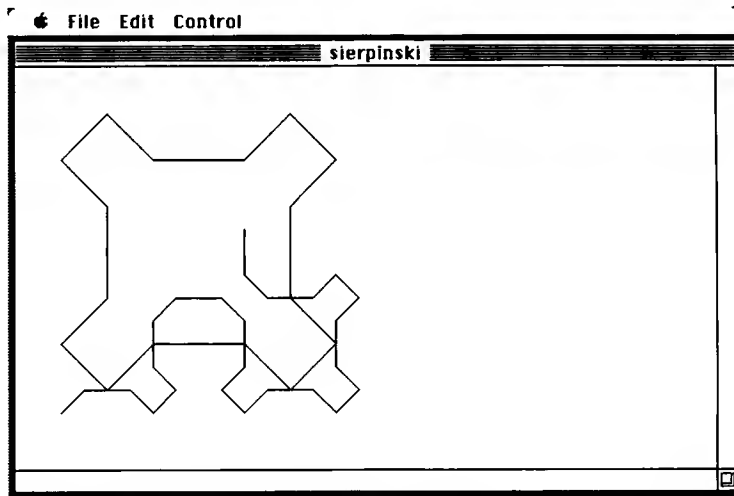


Illustration 10.16 Sierpinski in its first stages

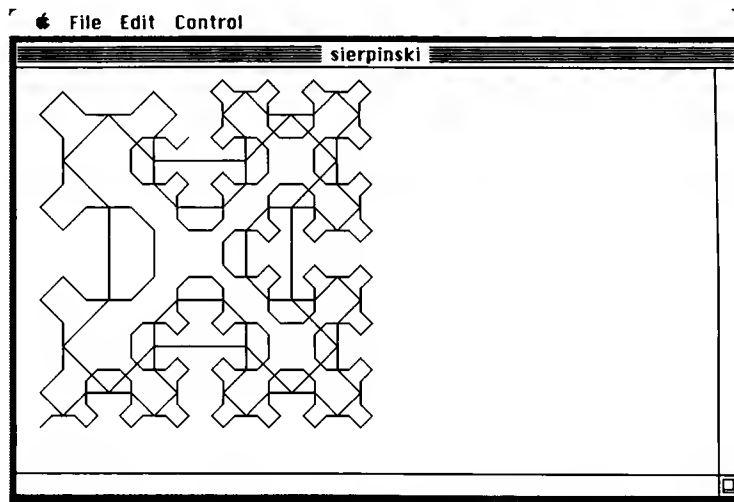


Illustration 10.17 Sierpinski some time later

As the program proceeds, the center design gets smaller and its corners grow not in size but in detail. This process produces an image called a *fractal*, which is a mathematically derived pattern that maintains the same level of detail no matter what the magnification of the image. We recommend to you the following sources if you are interested in this topic.

List of References on Fractals

- McGraw-Hill Yearbook on Science and Technology* (McGraw-Hill, Inc., New York) 1984, p. 191.
- Gannes, Stuart, "Lights, Cameras ... Computers", *Discover*, August 1984 (Time, Inc., Los Angeles), pp. 76-79.
- Mandelbrot, Benoit B., *The Fractal Geometry of Nature* (W. H. Freeman & Co., San Francisco, 1977).
- Mandelbrot, Benoit B., *Fractals, Form, Chance and Dimension* (W. H. Freeman & Co., San Francisco, 1977).
- McDermott, Jeanne, "Geometrical Forms Known as Fractals Find Sense in Chaos", p. 110, *Smithsonian Magazine* December 1983, p. 110.
- Sorensen, Peter R., "Simulating Reality with Computer Graphics", *Byte* 9 #3 (McGraw-Hill, Inc., N.H. March 1984) pp. 106-134.
- Tucker, Jonathan B., "Computer Graphics Achieve New Realism", *High Technology* 4 (High Technology Publishing Co., Los Angeles), June 1984, p. 42.
- Van Dam, Andries, "Computer Software for Graphics", *Scientific American* 251 #3 (Sept. 1984, Scientific American Inc., New York), pp. 146-159.

Centered Sierpinski

You may note that the Sierpinski curves above were oriented to the left of the screen. This is only because the Macintosh screen is roughly twice as wide as it is high. We can change the program slightly where it defines the width of the screen in line 100. The variable $H0$ refers to the screen width (or height, as Sierpinski's fill a square area) so change $H0=256$ to $H0=512$. With this small change, we can center the image, and double its size at the expense of not seeing the bottom half.

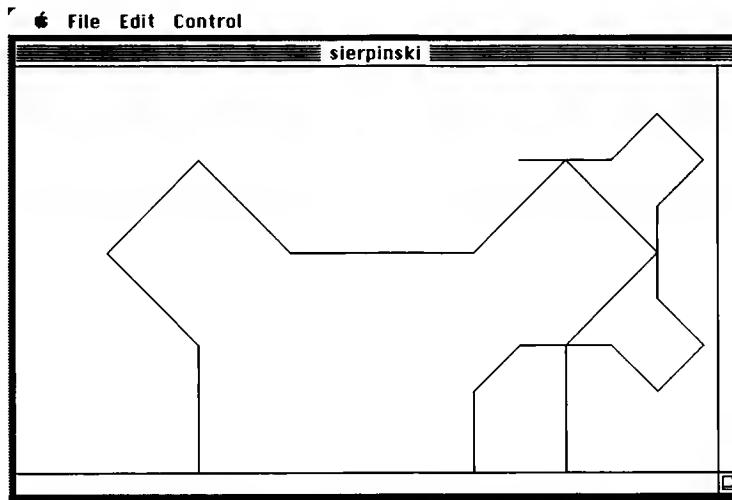


Illustration 10.18 Stage one of centered Sierpinski

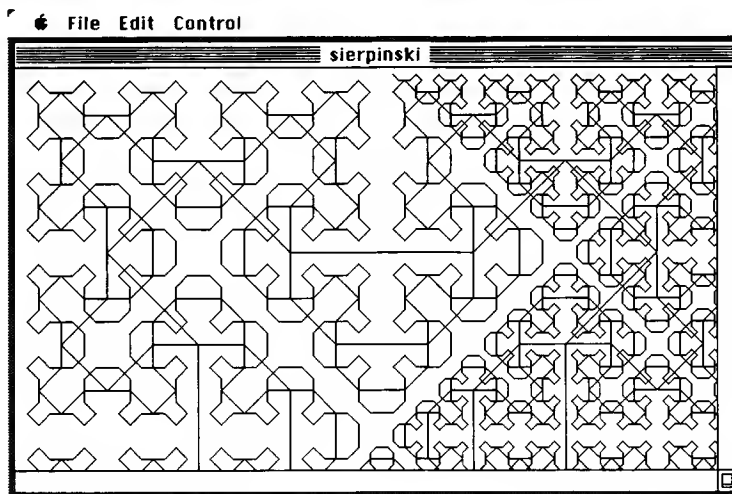


Illustration 10.19 Stage two

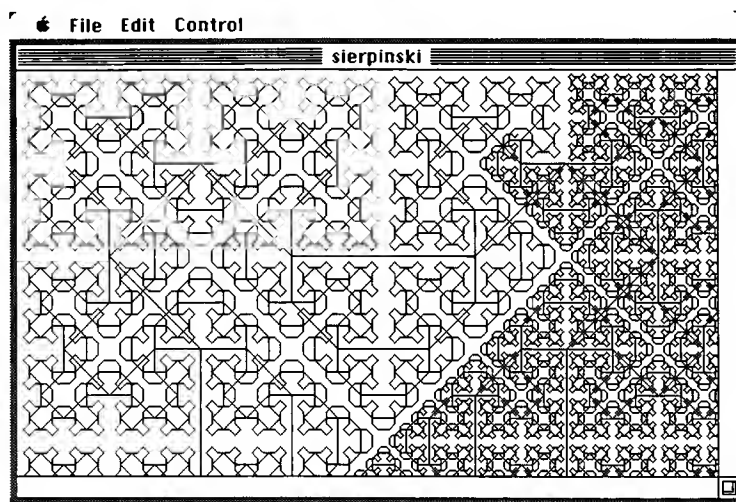


Illustration 10.20 Stage three

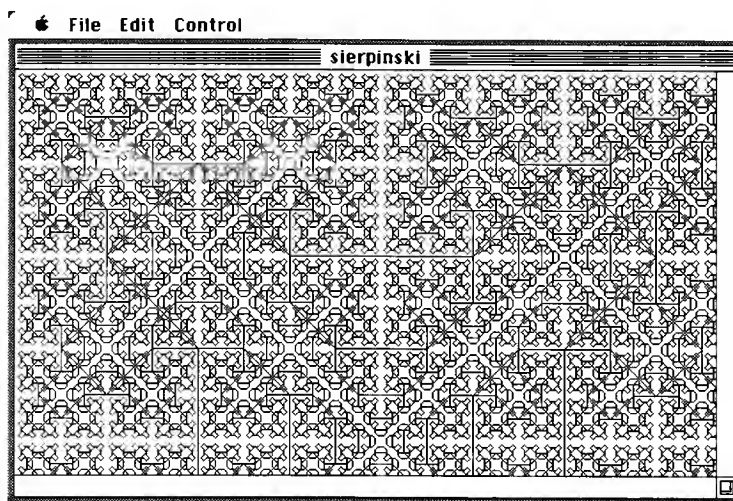


Illustration 10.21 Stage four

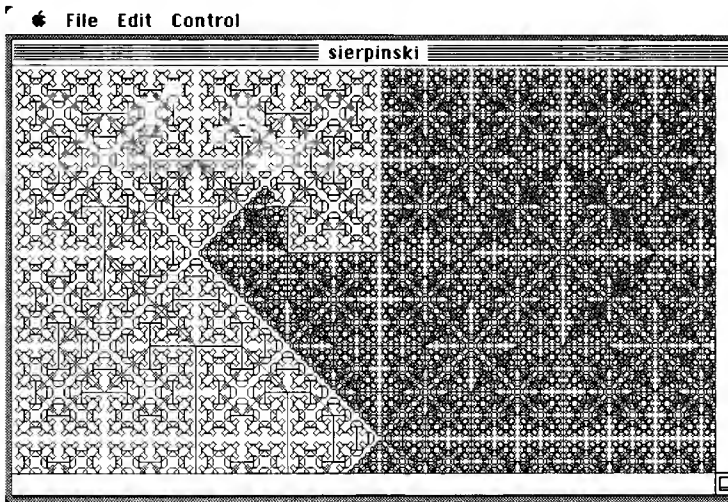


Illustration 10.22 Stage five

One of the intriguing aspects of these patterns is the different look of the patterns. The first ones with minimal detail seem somewhat boxy. Then the next set begin to take on the look of a tiled floor, which of course is the topic here, tessellation. The last patterns appear to be lace-like, with symmetry of course, but with a touch of roundedness that is caused by the miniscule squares that are drawn.

One last note about this series of images. Each major iteration in the program draws the top half on-screen and the bottom half off-screen. So the computer seems inactive for half of the time. This pause becomes quite long during the latter part of the program run because of the excruciatingly large number of lines the program must draw. Consider:

First image:	16 straight lines
Second image:	$12 + 4 * 15 = 72$ straight lines
Third image:	$12 + 4 * 72 = 300$ straight lines
Fourth image:	$12 + 4 * 300 = 1212$ straight lines
Fifth image:	$12 + 4 * 1212 = 4860$ straight lines
Sixth image:	$12 + 4 * 4860 = 19452$ straight lines
Seventh image:	$12 + 4 * 19452 = 77820$ straight lines
Eighth (last) image:	$12 + 4 * 77820 = 311292$ straight lines

All of this takes time, so you must have patience with this program. You will be rewarded with these fascinating images.

Bent Sierpinski

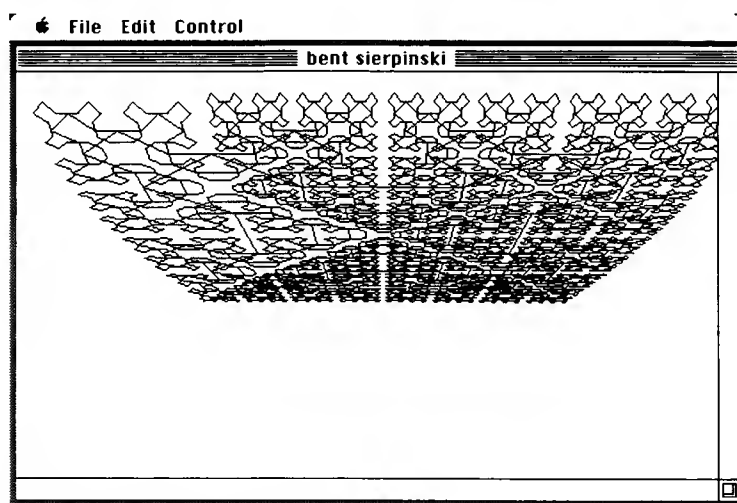


Illustration 10.23 Bent Sierpinski centered

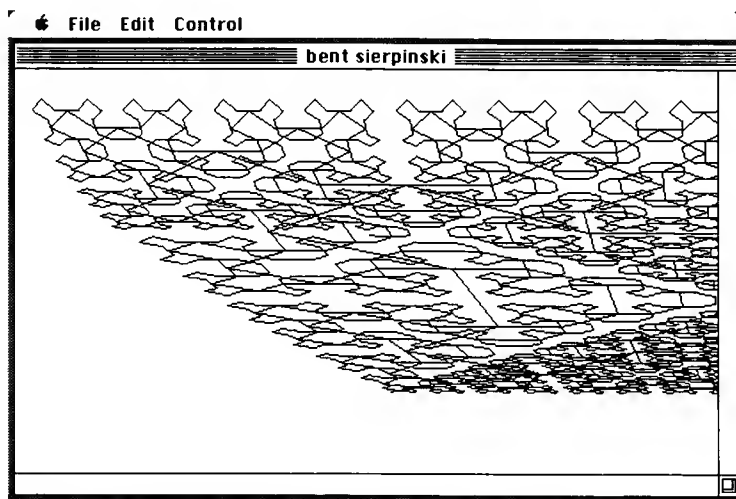


Illustration 10.24 Bent Sierpinski shifted right

The last two screen images you see in Illustrations 10.23 and 10.24 were produced with a minor modification to the program. Change the program this way:

Listing, Bent Sierpinski

```

10 ' bent sierpinski
15 ' Creative Computing, July 1984 v10, #7, pp.
    148-160 David H. Ahl
20 DEFINT A-Z
30 CLS
40 FOR DI=1 TO 7
60 GOSUB 100
80 NEXT DI
90 GOTO 90
100 H0=512: SP=0: H=H0/4: X=H+H: Y=X+H: I=0
110 I=I+1: X=X-H: H=H/2: Y=Y+H
115 YP=SQR(Y)*7
116 XP=X*(-Y/(H0+H0)+1)+Y/4
120 IF I<DI THEN 110
130 PS=I: GOSUB 600
140 GOSUB 200: A=H: B=-H: GOSUB 800
150 GOSUB 300: A=-H: B=-H: GOSUB 800
160 GOSUB 400: A=-H: B=H: GOSUB 800
170 GOSUB 500: A=H: B=H: GOSUB 800
180 GOSUB 700
190 RETURN
200 '
210 IF TP<=0 THEN RETURN
220 PS=TP-1: GOSUB 600
230 GOSUB 200: A=H: B=-H: GOSUB 800
240 GOSUB 300: A=H+H: B=0: GOSUB 800
250 GOSUB 500: A=H: B=H: GOSUB 800
260 GOSUB 200
270 GOSUB 700
280 RETURN
300 '
310 IF TP<=0 THEN RETURN
320 PS=TP-1: GOSUB 600
330 GOSUB 300: A=-H: B=-H: GOSUB 800
340 GOSUB 400: A=0: B=-2*H: GOSUB 800
350 GOSUB 200: A=H: B=-H: GOSUB 800
360 GOSUB 300
370 GOSUB 700
380 RETURN
400 '
410 IF TP<=0 THEN RETURN
420 PS=TP-1: GOSUB 600
430 GOSUB 400: A=-H: B=H: GOSUB 800
440 GOSUB 500: A=-2*H: B=0: GOSUB 800
450 GOSUB 300: A=-H: B=-H: GOSUB 800

```

(continued)

```
460 GOSUB 400
470 GOSUB 700
480 RETURN
500 '
510 IF TP<=0 THEN RETURN
520 PS=TP-1: GOSUB 600
530 GOSUB 500: A=H: B=H: GOSUB 800
540 GOSUB 200: A=0: B=H+H: GOSUB 800
550 GOSUB 400: A=-H: B=H: GOSUB 800
560 GOSUB 500
570 GOSUB 700
580 RETURN
600 '
610 SP=SP+1: ST(SP)=PS
620 TP=PS: RETURN
700 '
710 SP=SP-1: TP=ST(SP): RETURN
800 '
802 X=X+A: Y=Y+B: YQ=SQR(Y)*7
804 XQ=X*(-Y/(H0+H0)+1)+Y/4
806 LINE(XP,YP)-(XQ,YQ): XP=XQ: YP=YQ: RETURN
```

The effect is to produce images that seem to recede from view.

We now leave these remarkable tessellations to explore the syntax of the DRAW command and our DRAW subroutine in detail.

THE DRAW SUBROUTINE

The DRAW Command's Syntax

The DRAW command is different from most that you find in BASIC. It executes according to the way you want it to. You specify a string that describes the way you want the computer to draw, and DRAW that string. For example, if you have defined a string A\$ as a set of commands for the computer to execute with the DRAW command, you would write the line

```
DRAW A$
```

We can't rewrite BASIC for you, but we will provide you with a subroutine that will execute A\$ thus:

```
S$ = A$: GOSUB 1000
```

You place the string into a new variable called S\$, branch to the subroutine at line 1000, and EUREKA! the string is drawn.

When you define S\$, the string of commands to be drawn, you have available a wide variety of parameters that will make this a truly powerful subroutine. Illustration 11.1 provides an overview of the DRAW commands.

Motion Commands:

Command	Example	Action
Mx,y	"M250,200"	Move the draw position to specified X,Y coordinates
M \pm x, \pm y	"M + 20,-10"	Move <i>relative</i> to current position. Much like the LINE STEP command
Ud	"U20"	Move <i>up</i> a displacement of d pixels from current position
Dd	"D30"	Move <i>down</i> a displacement of d pixels
Ld	"L100"	Move <i>left</i> d pixels
Rd	"R40"	Move <i>right</i> d pixels
Ed	"E50"	Move <i>up and right</i> (think of New England)
Fd	"F20"	Move <i>down and right</i> (Florida)
Gd	"G3"	Move <i>down and left</i> (Gila monsters?)
Hd	"H20"	Move <i>up and left</i> (Mount St. Helens?)

Illustration 11.1 Table of motion commands for DRAW

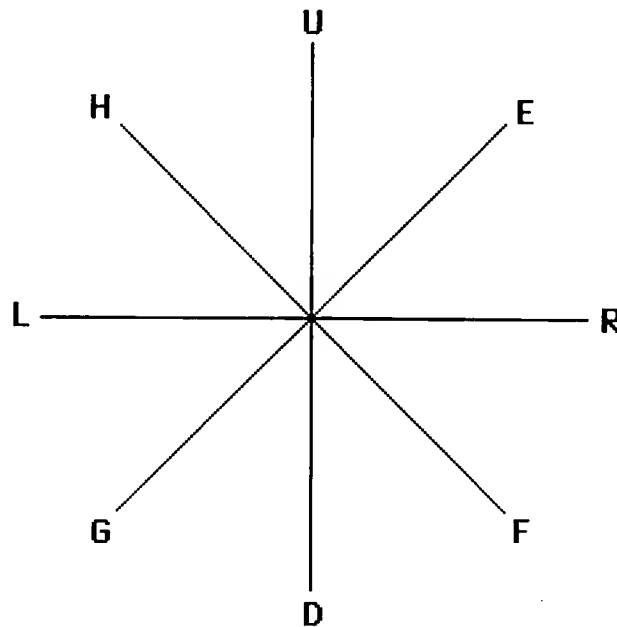


Illustration 11.2 Sketch of 8 directions for motion with DRAW

These MOVE commands can be combined into a string of several moves at a time. This is what makes the DRAW command so powerful, for otherwise you might as well do a LINE STEP(20,20) instead of a DRAW "F20".

Note: The LINE STEP(20,20) and DRAW "F20" are equivalent, even though you might think that the DRAW command would draw a line 20 pixels long. It doesn't. It draws a diagonal line whose horizontal and vertical components are each 20 pixels long. The actual length is

$$r = \sqrt{20^2 + 20^2} = 28.28$$

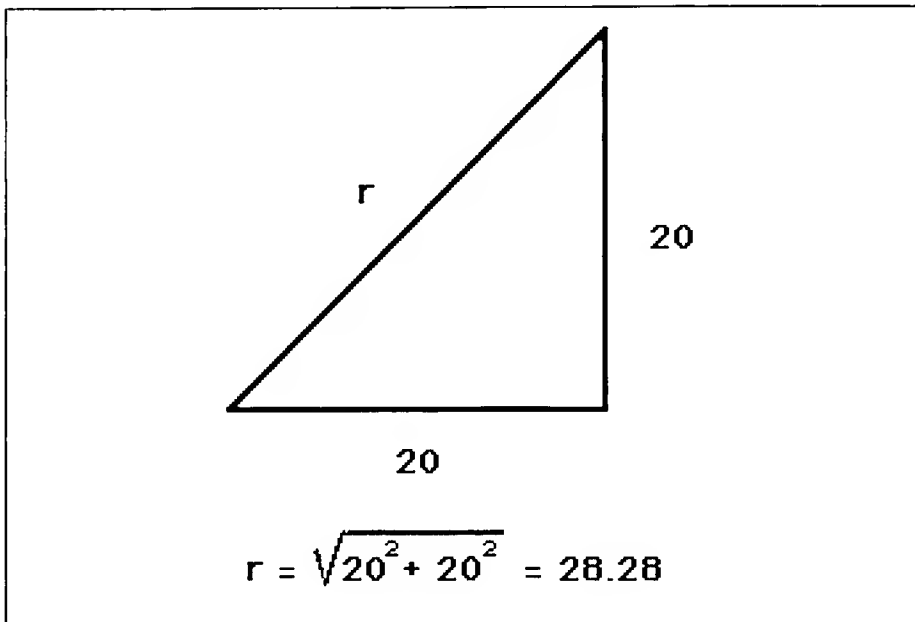


Illustration 11.3 Triangle with diagonal measured as the square root of sum of squares (Pythagorean theorem)

To combine several MOVE instructions within a single string for the DRAW command to execute, simply enter them one after the other, and separate them (if you desire, as it is optional) with a semicolon.

Examples:

DRAW "R40;D70;L40;U70"

A rectangle is drawn with the longer side vertical.

DRAW "M250,200;E40;F40;G40;H40"

A diamond is drawn down from the center of the screen.

```
DRAW "M0,0;R100D100L100U100"
```

A square at the top left of the screen.

```
FOR I=1 TO 100
  DRAW "+2,-1"
NEXT I
```

A 30-degree diagonal up and to the right.

Options:

Command	Example	Action
B	"BM0,0"	Blank move. Like a "Lift up pen"
N	"R20ND20"	No position change. This example draws from origin right 20, then from origin down 20.
X	A\$ = "BM250,220" DRAW "L25U25; XA\$"	Execute the substring defined in the DRAW. This example draws left 25, up 25, then executes the command A\$. We did not implement the X option when we wrote the DRAW subroutine.

Illustration 11.4 Options of DRAW command

Modes:

Mode	Example	Action
Ax	"R20;A1; R20;A2; R20;A3;R20"	Changes angle of all subsequent draws. x=0, no change x=1, 90 degrees clockwise x=2, 180 degrees clockwise x=3, 270 degrees clockwise In the example above, a line is drawn right, then down, then left, then up — even though the command suggests all lines go right. We did not implement the A mode.
Cc	"C33;R20"	Color is 33. This command is not used in the DRAW subroutine as we wrote it, as it is generally useful only on color systems.

Sx	"S2;R20; S4;R20"	Scale, with x a number from 1 to 62. x = 1, 1/4 scale x = 2, 2/4 scale (1/2 scale) x = 3, 3/4 scale x = 4, full scale x = 8, twice scale
----	---------------------	---

Illustration 11.5 Modes of the DRAW command

DRAW Subroutine

In order to clarify the next tessellation with the DRAW subroutine, we will describe the subroutine in pseudocode. The pseudocode listing will be keyed to line numbers in the subroutine you will find on pages 166 and 167 as part of the program ANGLEDRAW, so that you can follow the logic of the program as you trace the instructions.

1010	1.	Initialize flags if F = 0 (first time called). B = Blank Move flag. Set B = 0 (draw). N = No Position Change flag. Set N = 1 (change). S = Scale. Set S = 4 (full scale). Q = scale multiplier. Set Q = 1 (full scale). F = First time called. Set F = 1 (first time).
1020	2.	Set character (char.) counter I8 to 1.
1030	3.	WHILE I8th. char. in S\$ is not a "#" DO:
1040		a. Call I8th. char. S8\$.
1050		b. IF char. is B then set B = 0.
1060		c. IF char. is N then set N = 0.
1070		d. IF char. is a directed move (U,D,L,R,E,F,G, or H) then PERFORM Process Directed Move.
1080		e. IF char. is M then PERFORM Process Move.
1090		f. IF char. is S then PERFORM Process Scale.
1100		g. Add 1 to char. counter I8.
1110	4.	ENDDO.
1120	5.	RETURN

Process Move

1210	1.	PERFORM <i>Pick up digits, sign if any</i> routine at 1900.
1220	2.	Save B8\$ in X8\$ Add 1 to I8 PERFORM <i>Pick up digits, sign if any</i> routine at 1900.
1230	3.	Save B8\$ in Y8\$ Set X8 to be value of X8\$ times Q, the scale Set Y8 to be value of Y8\$ times Q
1240	4.	IF first char. in X8\$ is "+" or "-" THEN a. $X = X + X8$ b. $Y = Y + Y8$ ELSE a. $X = X8$ b. $Y = Y8$
1250	9.	IF B = 1 (not a blank move) THEN draw line to (X,Y) ELSE move cursor to (X,Y)
1260	10.	RETURN

Process Directed Move

1310	1.	Set X8 and Y8 = 0 PERFORM <i>pick up digits, sign if any</i> at 1900.
1320	2.	Set B8 to value of B8\$ Set P8 to position of char. S8\$ in string "LRUDEFHG"
1330	3.	IF P8 = 1 or P8 > 6 THEN X8 = - B8 (S8\$ is "L", "G", or "H")
1340	4.	IF P8 = 2 or P8 = 5 or P8 = 6 THEN X8 = B8 (S8\$ is "R", "E", or "F")
1350	5.	IF P8 = 3 or P8 = 5 or P8 = 8 THEN Y8 = - B8 (S8\$ is "U", "E", or "H")
1360	6.	IF P8 = 4 or P8 = 6 or P8 = 7 THEN Y8 = B8 (S8\$ is "D", "F", or "G")
1370	7.	IF X + X8 out of bounds THEN X8 = - X8
1380	8.	IF Y + Y8 out of bounds THEN Y8 = - Y8
1390	9.	Draw line to (X8,Y8)
1400	10.	IF N = 1 THEN a. $X = X + X8$ b. $Y = Y + Y8$ ELSE a. Move to (X,Y) b. Set N = 1
1410	11.	RETURN

Process Pick up Sign, Digits if any

```

1910  1.      Set B8$ = null.
1920  2.      WHILE l8th char. is one of chars. in "0123456789 - +" DO:
1930          Add to B8$ the character found in S$
          Add 1 to l8
1940  3.      ENDDO.
1950  RETURN

```

Applications of DRAW Subroutine

The first program that uses the DRAW is very short. It provides the same output as the first tessellation program that was demonstrated in this chapter.

Remember that in that program we selected a 10-unit-long line to be drawn in a random vertical, horizontal, or diagonal direction from where we were. The result was a trace of a random walk, which when allowed to proceed for some time ended up as a tessellation of the screen (Illustration 11.6 and 11.7).

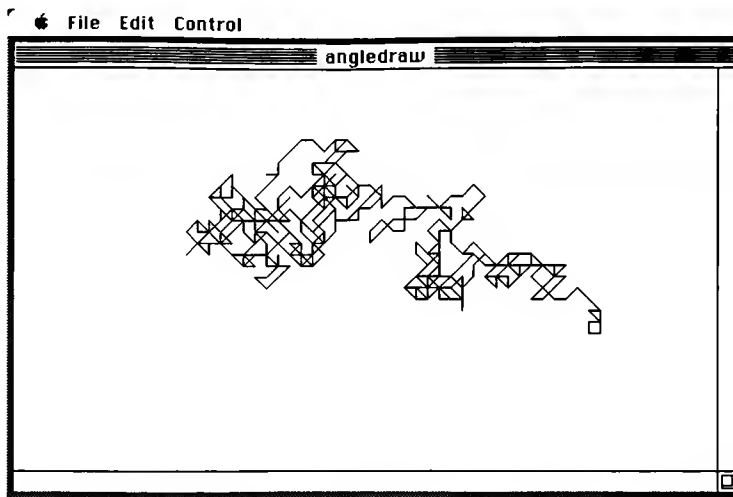


Illustration 11.6 Angle draw

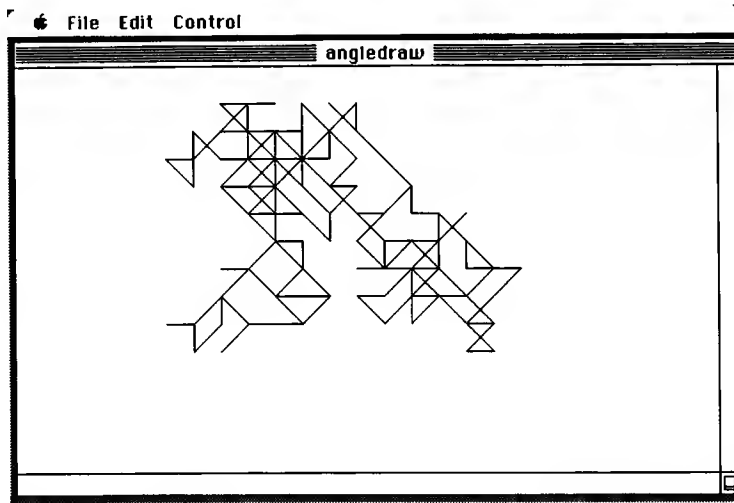


Illustration 11.7 Angle draw, longer segments

Here we do the same thing, only we use the Directed Move portion of the DRAW subroutine to select the direction of our drunkard's next step. Study the listing.

Listing, ANGLEDRAW Program

```

10 ' angledraw
20 CLS: RANDOMIZE TIMER: F=0
30 S$="BM200,120S3#":GOSUB 1000
40 S$=MID$("UDLREFGH",1+8*RND,1)+"10#":GOSUB 1000
50 GOTO 40
1000 ' DRAW subroutine
1010 IF F=0 THEN B=1:N=1:S=4:Q=1:F=1
1020 I8=1
1030 WHILE MID$(S$,I8,1)<>"#"
1040 S8$=MID$(S$,I8,1)
1050 IF S8$="B" THEN B=0: GOTO 1100
1060 IF S8$="N" THEN N=0: GOTO 1100
1070 IF INSTR("UDLREFGH",S8$)<>0 THEN GOSUB 1300:
      GOTO 1100
1080 IF S8$="M" THEN GOSUB 1200: GOTO 1100
1090 IF S8$="S" THEN GOSUB 1900: S=VAL(B8$): Q=S/4:
      GOTO 1100
1100 I8=I8+1
1110 WEND
1120 RETURN
1200 ' process M
1210 GOSUB 1900 '>>>> get digits, sign for X

```

```

1220 X8$=B8$: I8=I8+1: GOSUB 1900 '>>>> bypass
      comma, get digits, sign for Y
1230 Y8$=B8$: X8=Q*VAL(X8$):Y8=Q*VAL(Y8$)
1240 IF INSTR("+-", LEFT$(X8$,1))<>0 THEN
      X=X+X8:Y=Y+Y8 ELSE X=X8:Y=Y8
1250 IF B=1 THEN CALL LINETO(X,Y) ELSE CALL
      MOVETO(X,Y):B=1
1260 RETURN
1300 'process directed move
1310 X8=0: Y8=0: GOSUB 1900 '>>>> get digits
1320 B8=Q*VAL(B8$): P8=INSTR("LRUDEFGH",S8$)
1330 IF P8=1 OR P8>6 THEN X8=-B8
1340 IF P8=2 OR P8=5 OR P8=6 THEN X8=B8
1350 IF P8=3 OR P8=5 OR P8=8 THEN Y8=-B8
1360 IF P8=4 OR P8=6 OR P8=7 THEN Y8=B8
1370 IF X+X8<0 OR X+X8>500 THEN X8=-X8
1380 IF Y+Y8<0 OR Y+Y8>280 THEN Y8=-Y8
1390 CALL LINE(X8,Y8)
1400 IF N=1 THEN X=X+X8: Y=Y+Y8 ELSE CALL
      MOVETO(X,Y): N=1
1410 RETURN
1900 'pick up digits, sign if any
1910 B8$=""
1920 WHILE INSTR("0123456789-+",MID$(S$,I8+1,1))<>0
1930 B8$=B8$+MID$(S$,I8+1,1): I8=I8+1
1940 WEND
1950 RETURN
9999 END

```

- 20 Clears the screen, seeds the random number generator, and sets F, the First Time Flag that DRAW needs, to 0.
- 30 Establish the starting point of our tessellation at (200,120), and scale at 3/4.
- 40 Determine the direction of the drunkard's random step by choosing the Move direction at random from the string "UDLREFGH". Then it sets the step size as 10. The DRAW string is terminated with a "#", and the DRAW subroutine is invoked.
- 50 Loop back to Line 40 to provide us with an infinite loop.

You can have great fun with this very simple program. We suggest that you start by altering line 40 in each of several ways:

```

40 S$=MID$("LLRRUDFH" .....
40 S$=MID$("UUDDLREG" .....
40 S$=MID$("EFGHEFGH" .....
40 S$=MID$("LRUDLRUD" .....

```

You can also tessellate the screen by selecting a random square area on the screen to fill with patterns of your choice. Subdivide the screen as a grid of squares, or rectangles, as shown in Illustration 11.8:

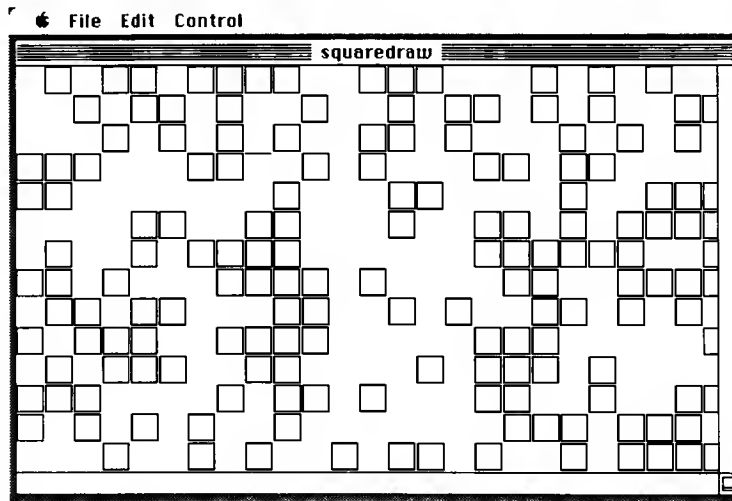


Illustration 11.8 Square tessellation with DRAW

Listing, SQUAREDRAW

```

10 ' squaredraw
20 CLS: RANOOIMIZE TIMER: F=0
30 R1=INT(25*RND)*20
40 R2=INT(14*RND)*20
50 R1$=MID$(STR$(R1),2): R2$=MID$(STR$(R2),2)
60 S$="BM"+R1$+"", "+R2$
70 S$=S$+"R18018L18U18#"
80 GOSUB 1000
90 GOTO 30
1000 ' DRAW subroutine
1010 IF F=0 THEN B=1:N=1:S=4:Q=1:F=1
1020 I8=1
1030 WHILE MID$(S$,I8,1)<>"#"
1040 S8$=MID$(S$,I8,1)
1050 IF S8$="B" THEN B=0: GOTO 1100
1060 IF S8$="N" THEN N=0: GOTO 1100
1070 IF INSTR("UOLREFGH",S8$)<>0 THEN GOSUB 1300:
      GOTO 1100
1080 IF S8$="M" THEN GOSUB 1200: GOTO 1100
1090 IF S8$="S" THEN GOSUB 1900: S=VAL(B8$): Q=S/4:
      GOTO 1100
1100 I8=I8+1

```

```

1110 WEND
1120 RETURN
1200 ' process M
1210 GOSUB 1900 '>>>> get digits, sign for X
1220 X8$=B8$: I8=I8+1: GOSUB 1900 '>>>> bypass
      comma, get digits, sign for Y
1230 Y8$=B8$: X8=Q*VAL(X8$):Y8=Q*VAL(Y8$)
1240 IF INSTR("+-", LEFT$(X8$,1))<>0 THEN
      X=X+X8:Y=Y+Y8 ELSE X=X8:Y=Y8
1250 IF B=1 THEN CALL LINETO(X,Y) ELSE CALL
      MOVETO(X,Y):B=1
1260 RETURN
1300 'process directed move
1310 X8=0: Y8=0: GOSUB 1900 '>>>> get digits
1320 B8=Q*VAL(B8$): P8=INSTR("LRUDEF GH",S8$)
1330 IF P8=1 OR P8>6 THEN X8=-B8
1340 IF P8=2 OR P8=5 OR P8=6 THEN X8=B8
1350 IF P8=3 OR P8=5 OR P8=8 THEN Y8=-B8
1360 IF P8=4 OR P8=6 OR P8=7 THEN Y8=B8
1370 IF X+X8<0 OR X+X8>500 THEN X8=-X8
1380 IF Y+Y8<0 OR Y+Y8>280 THEN Y8=-Y8
1390 CALL LINE(X8,Y8)
1400 IF N=1 THEN X=X+X8: Y=Y+Y8 ELSE CALL
      MOVETO(X,Y): N=1
1410 RETURN
1900 'pick up digits, sign if any
1910 B8$=""
1920 WHILE INSTR("0123456789-+",MID$(S$,I8+1,1))<>0
1930 B8$=B8$+MID$(S$,I8+1,1): I8=I8+1
1940 WEND
1950 RETURN
9999 END

```

You can do this by selecting the upper left corner of a random grid area this way:

```

100 'get random square grid area
110 'horiz. coord. R1 is 0, 20, 40, 60, ... to 500
120 R1=INT(25*RND)*20
130 'vert. coord. R2 is 0, 20, 40, 60, ... to 280
140 R2=INT(14*RND)*20
160 'String representation of R1 and R2 for DRAW:
170 R1$=MID$(STR$(R1),2): R2$=MID$(STR$(R2),2)

```

Once you have established the random corner coordinates, you can issue a DRAW command to place your pattern in that area. For example, suppose you want to draw a diamond with a vertical and horizontal line within it, as shown in Illustration 11.9:

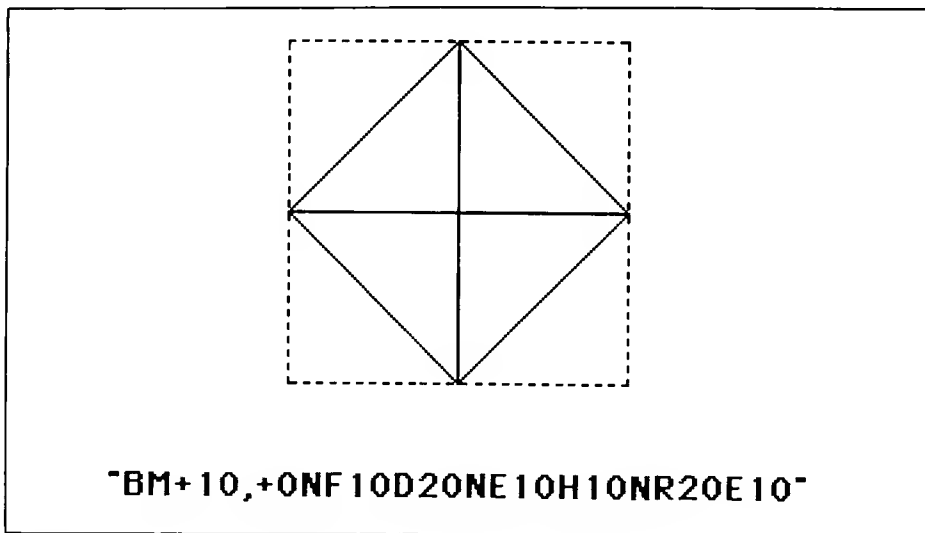


Illustration 11.9 Diamond with vertical and horizontal lines

To the code above, you add these lines:

```
180 S$="BM"+R1$+","+R2$ 'move to random corner
190 S$=S$+"BM+10,+0NF10D20NE10H10NR20E10#"
200 GOSUB 1000
```

Line 190 defines the DRAW command as

Blank move right 10 from present position.

Draw down and right 10, go back to top of diamond.

Draw down 20.

Draw up and right 10, go back to bottom of diamond.

Draw up and left 10.

Draw right 20, go back to left point of diamond.

Draw up and right 10, return for next DRAW.

You can define your tile area as a rectangle by modifying line 120 as shown below:

```
120 R1=INT(20*RND)*25)
```

This allows 20 rectangles across the screen, each 25 units wide. Now you can define a new pattern in this different tile.

Alphabet Generator

The last application we have chosen to demonstrate using the DRAW command is an alphabet generator. What we do in this program is to define 37 different DRAW commands as DATA strings, representing the motions necessary to trace the 26 letters of the alphabet, the 10 digits, and the space character. When these 37 strings are stored in an array, the program can select them by position and execute them. The commands to draw the A are stored as a string in the first array position; those to draw the B in the second consecutively to the Z in the 26th. Thus to draw an A, place the contents of the first array position into S\$, and call the DRAW subroutine.

position

A\$

1	Command string to draw "A"
2	Command string to draw "B"
3	Command string to draw "C"
4	Command string to draw "D"
5	Command string to draw "E"
6	Command string to draw "F"
7	Command string to draw "G"
8	Command string to draw "H"
9	Command string to draw "I"
0	Command string to draw "J"
11	Command string to draw "K"
12	Command string to draw "L"
13	Command string to draw "M"
14	Command string to draw "N"
15	Command string to draw "O"
16	Command string to draw "P"
17	Command string to draw "Q"
18	Command string to draw "R"
19	Command string to draw "S"

20	Command string to draw "T"
21	Command string to draw "U"
22	Command string to draw "V"
23	Command string to draw "W"
24	Command string to draw "X"
25	Command string to draw "Y"
26	Command string to draw "Z"
27	Command string to draw "0"
28	Command string to draw "1"
29	Command string to draw "2"
30	Command string to draw "3"
31	Command string to draw "4"
32	Command string to draw "5"
33	Command string to draw "6"
34	Command string to draw "7"
35	Command string to draw "8"
36	Command string to draw "9"
37	Command string to draw " "

Illustration 11.10 Sketch of array with commands stored

Here's the listing of the program that allows you to write (to DRAW) words in a new font.

Listing, ALPHABET DRAW

```

10 ' alphabet draw
20 DIM A$(37)
30 CLS:CALL MOVETO(10,10):X=10:Y=10:F=0
40 FOR I=1 TO 37
50 READ A$(I)
60 NEXT I
70 DATA "BM+0,+20E12D8NL8D4BM+8,-20"
80 DATA "BM+0,+8R12D12L12U12U8BM+20,+0"
90 DATA "BM+12,+20L12U12NR12BM+20,-8"
100 DATA "BM+0,+8R12D12L12U12BM+12,+0U8BM+8,+0"
110 DATA "BM+12,+20L12U12NR12BM+0,+6R12BM+8,-14"
120 DATA "BM+0,+8NR12D12U6R8BM+12,-14"
130 DATA "BM+0,+8R12D12L12U12BM+12,+12D8L6BM+14,-
    28"
140 DATA "BM+12,+20U12L12ND12U8BM+20,+0"
150 DATA "BM+6,+8D12BM+6,+0L12BM+6,-18D2BM+14,-4"
160 DATA "BM+12,+8D20L12U8BM+12,-18D2BM+8,-4"
170 DATA "D20U8NE12E2F10BM+8,-20"
180 DATA "D20R12BM+8,-20"
190 DATA "BM+12,+20U12L12ND12BM+6,+0D12BM+14,-20"
200 DATA "BM+12,+20U12L12ND12BM+20,-8"
210 DATA "BM+0,+8R12D12L12U12BM+20,-8"
220 DATA "BM+0,+8R12D12L12U12D20BM+20,-28"
230 DATA "BM+0,+8R12D12L12U12BM+12,+0D20BM+8,-28"
240 DATA "BM+12,+8L12D12BM+20,-20"
250 DATA "BM+12,+8L12D6R12D6L12BM+20,-20"
260 DATA "BM+0,+8R6ND12R6BM+8,-8"
270 DATA "BM+12,+8D12L12U12BM+20,-8"
280 DATA "BM+0,+8D12E12BM+8,-8"
290 DATA "BM+12,+8012L12U12BM+6,+0D12BM+14,-20"
300 DATA "BM+6,+14NE6NF6NG6NH6BM+14,-14"
310 DATA "BM+12,+8D12L12U12BM+6,+12D8BM+14,-28"
320 DATA "BM+0,+8R12G12R12BM+8,-20"
330 DATA "D20R12U20L12BM+20,+0"
340 DATA "BM+6,+0NG4D20NR6L6BM+20,-20"
350 DATA "R12D10L12D10R12BM+8,-20"
360 DATA "R12D10NL12D10L12BM+20,-20"
370 DATA "D10R12NU10D10BM+8,-20"
380 DATA "NR12D10R12D10L12BM+20,-20"
390 DATA "NR4D20R12U10L12BM+20,-10"
400 DATA "R12D20BM+8,-20"
410 DATA "ND20R12D10NL12D10L12BM+20,-20"
420 DATA "ND10R12D10NL12D10BM+8,-20"
430 DATA "BM+20,+0"

```

```

440 S$=INKEY$:IF S$="" THEN 440
450 IF S$<>CHR$(13) THEN 480
460 X=10:Y=Y+36*Q: S$="BM"+STR$(X)+", "+STR$(Y)+"#"
470 GOSUB 1000: GOTO 440
480 IF S$="/" THEN STOP
490 J=INSTR("abcdefghijklmnopqrstuvwxy0123456789",S$)
500 IF J=0 THEN J=37
510 S$=A$(J)
520 S$="S16"+S$+"#"
530 GOSUB 1000
540 GOTO 440
1000 ' ORAW subroutine
1010 IF F=0 THEN B=1:N=1:S=4:Q=1:F=1
1020 I8=1
1030 WHILE MIO$(S$,I8,1)<>"#"
1040 S8$=MIO$(S$,I8,1)
1050 IF S8$="B" THEN B=0: GOTO 1100
1060 IF S8$="N" THEN N=0: GOTO 1100
1070 IF INSTR("UOLREFGH",S8$)<>0 THEN GOSUB 1200:
      GOTO 1100
1080 IF S8$="M" THEN GOSUB 1130: GOTO 1100
1090 IF S8$="S" THEN GOSUB 1320: S=VAL(B8$): Q=S/4:
      GOTO 1100
1100 I8=I8+1
1110 WEND
1120 RETURN
1130 ' process M
1140 GOSUB 1320 '>>>> get digits, sign for X
1150 X8$=B8$: I8=I8+1: GOSUB 1320 '>>>> bypass
      comma, get digits, sign for Y
1160 Y8$=B8$: X8=Q*VAL(X8$):Y8=Q*VAL(Y8$)
1170 IF INSTR("+-",LEFT$(X8$,1))<>0 THEN
      X=X+X8:Y=Y+Y8 ELSE X=X8:Y=Y8
1180 IF B=1 THEN CALL LINETO(X,Y) ELSE CALL
      MOVETO(X,Y):B=1
1190 RETURN
1200 'process directed move
1210 X8=0: Y8=0: GOSUB 1320 '>>>> get digits
1220 B8=Q*VAL(B8$): P8=INSTR("LRUOEFGH",S8$)
1230 IF P8=1 OR P8>6 THEN X8=-B8
1240 IF P8=2 OR P8=5 OR P8=6 THEN X8=B8
1250 IF P8=3 OR P8=5 OR P8=8 THEN Y8=-B8
1260 IF P8=4 OR P8=6 OR P8=7 THEN Y8=B8
1270 IF X+X8<0 OR X+X8>500 THEN X8=-X8
1280 IF Y+Y8<0 OR Y+Y8>280 THEN Y8=-Y8
1290 CALL LINE(X8,Y8)
1300 IF N=1 THEN X=X+X8: Y=Y+Y8 ELSE CALL
      MOVETO(X,Y): N=1

```

(continued)

```
1310 RETURN
1320 'pick up digits, sign if any
1330 B8$=""
1340 WHILE INSTR("0123456789-+",MID$(S$,18+1,1))<>0
1350 B8$=B8$+MID$(S$,18+1,1): 18=18+1
1360 WEND
1370 RETURN
9999 END
```

20	Define array A\$ to hold 37 command strings.
30	Clear screen, move cursor to upper left corner, set X and Y starting points to that cursor position, set First Time Flag F to 0.
40-60	Read command strings into array A\$.
70-430	Commands themselves. Consider line 70 as a typical example, defining the letter A. It states: "Blank Move (relative to current position) 20 down (go to bottom left of 20-by-20 square), go up and right 12 units" (Illustration 11.11). "Go down 8 units, left 8 units, return to previous position" (Illustration 11.12). "Go down 4 units to finish off the A" (Illustration 11.13). "Blank Move right 8, up 20 to top right of this drawing area, which is top left of next area."
440	Pick up a keystroke from user.
450-480	IF char. = "/" then STOP ELSE return cursor to new line, 10 over and 36*Q down.
490	Place in J position of S\$ in array, if it is one of the 37.
500	Define character as a blank if it is not one of the 37 acceptable characters.
510	Place in S\$ the appropriate command string from the array A\$.
520	Set scale to 16 and place terminator symbol at end of S\$.
530	Perform DRAW subroutine.
540	Loop back to pick up another character.

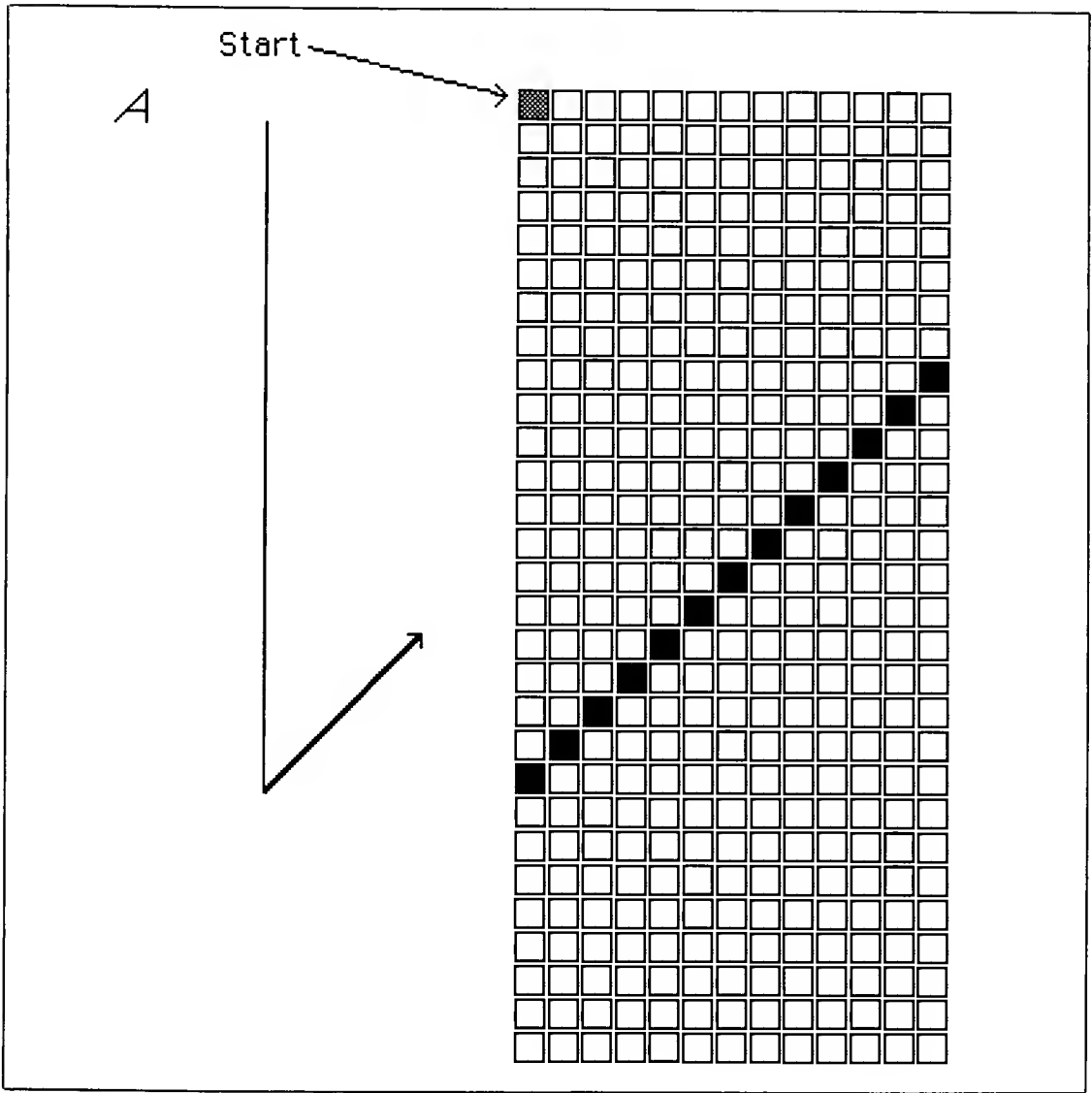


Illustration 11.11 Diagonal bar of A in rectangle

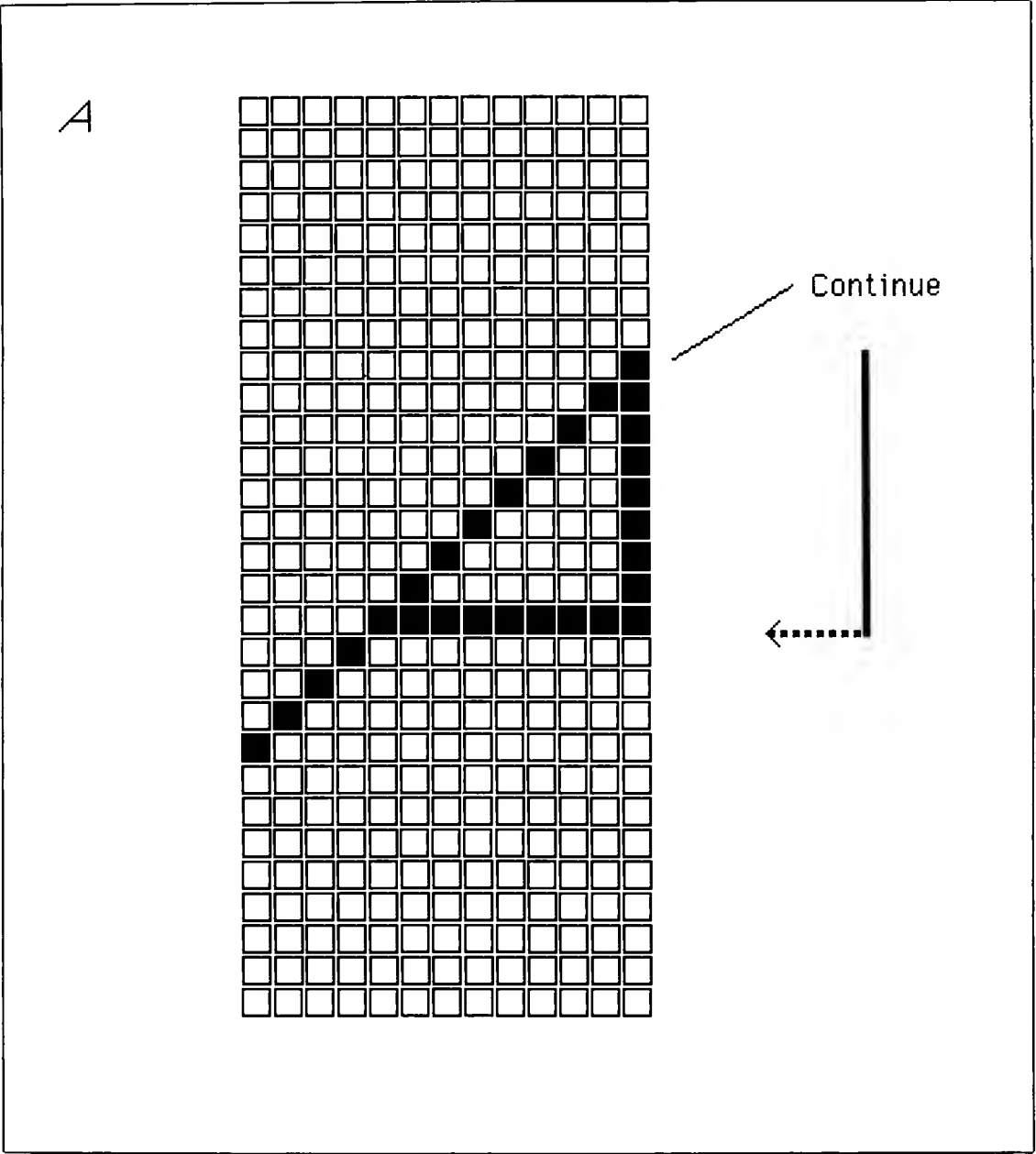


Illustration 11.12 Letter A later

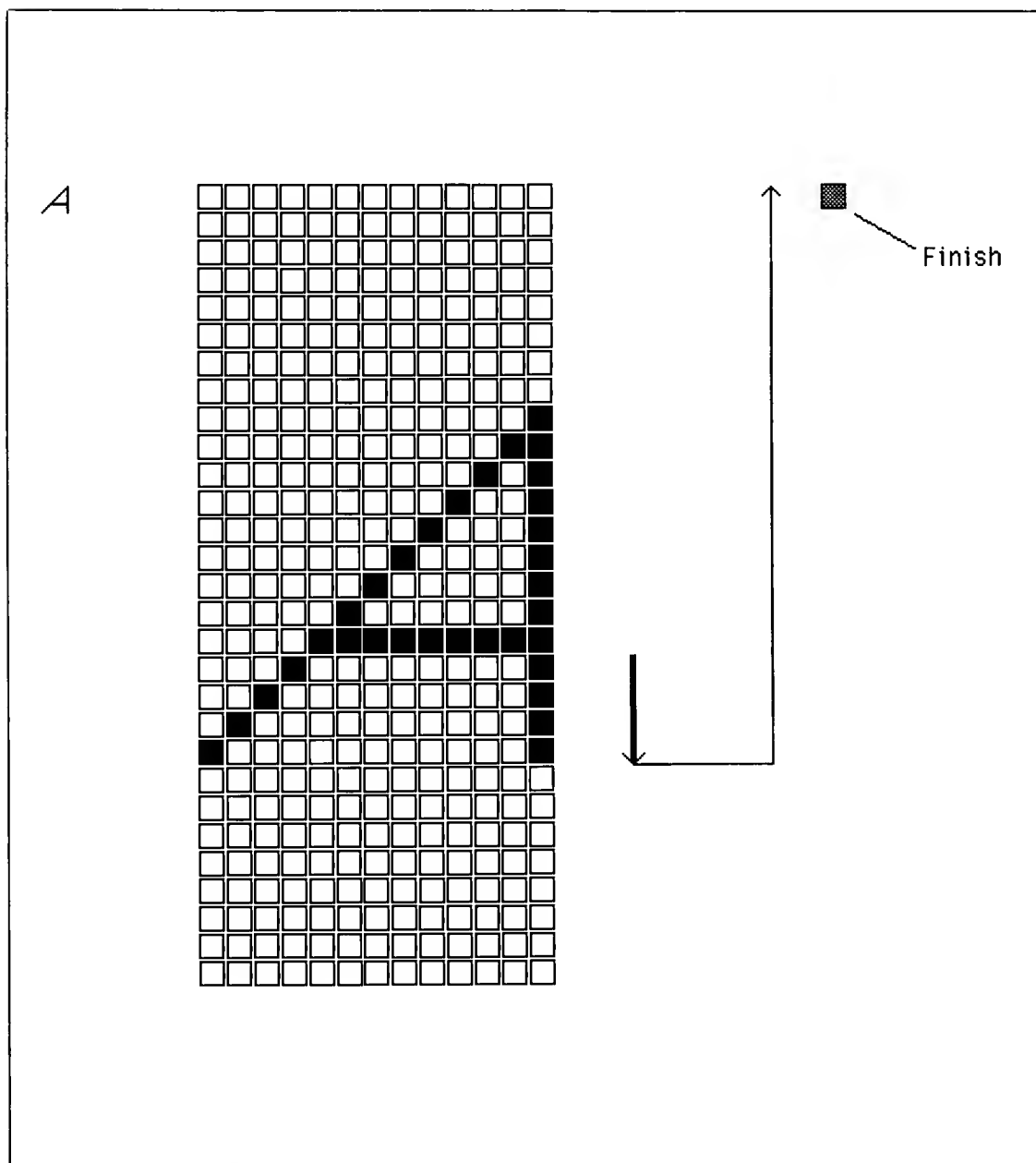


Illustration 11.13 Finished letter A

The screen images below (Illustration 11.14 - 11.17) indicate some of the flexibility of this program. Of course you are free to define your own fonts by rewriting the command strings in lines 70-430. That's half the fun of this program.

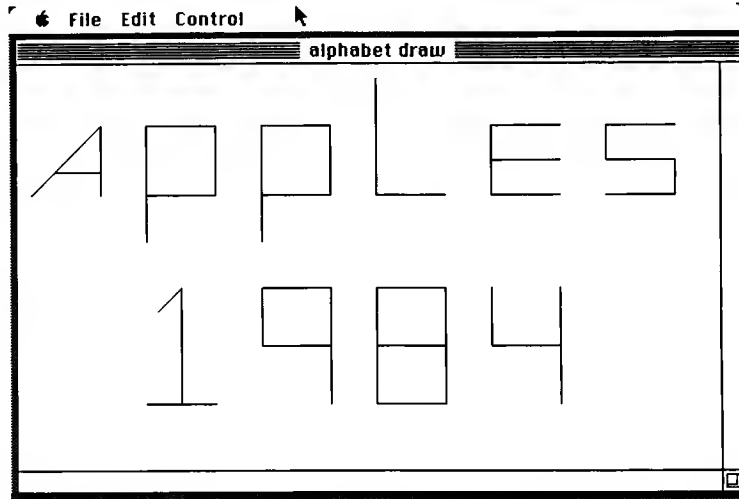


Illustration 11.14 The Year of the Apple

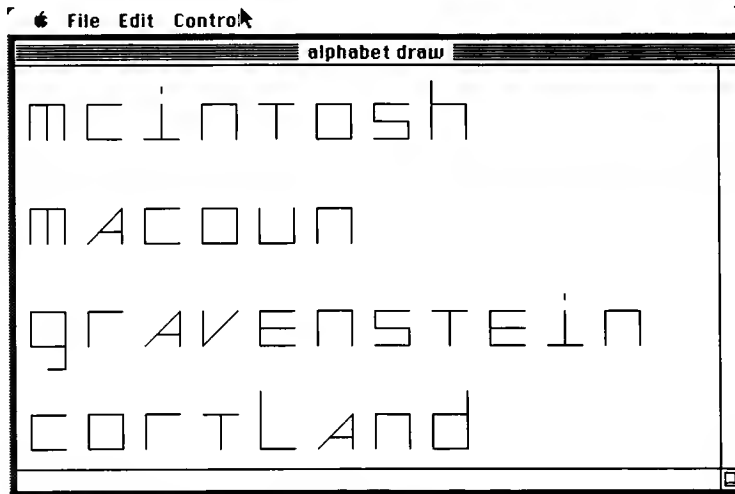


Illustration 11.15 An Apple by any other name

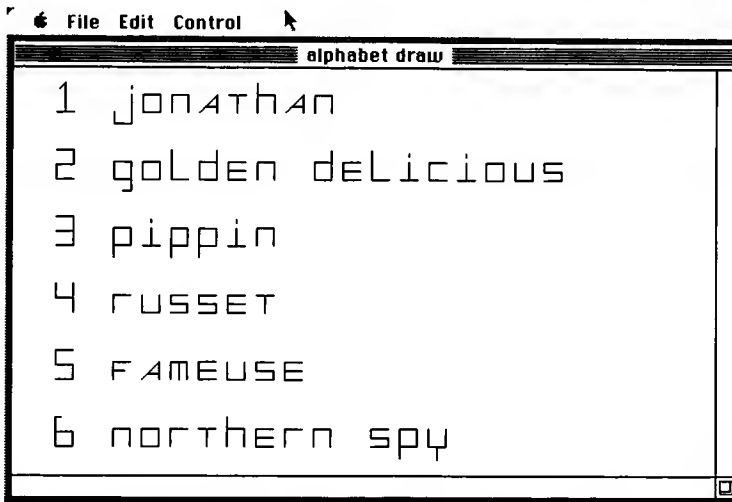


Illustration 11.16 An Apple a day, never on Sunday

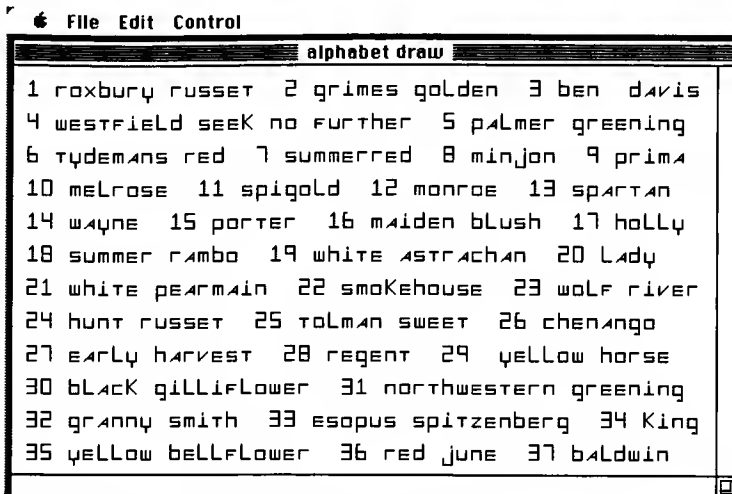


Illustration 11.17 How 'bout them apples?

This chapter has demonstrated several features of the Macintosh graphics that you can create in BASIC. As usual, you are encouraged to play with these ideas, and to expand on the possibilities.

MOUSE TANS

We revisit the topic of tangrams in this chapter, though in a very different guise. In Chapter Seven the subject was tangrams produced by manipulating figures in MacPaint. Here we used MacPaint tools such as the lasso, the marquee, and the *Edit menu's* powerful *Flip vertical* and *Flip horizontal*. We will describe, and discuss in detail, a program in BASIC that uses the mouse to select and execute commands to draw the tangram.

Illustration 12.1 shows how the screen output area of BASIC was subdivided into four parts, each with its own function. The four area labels, *selection square*, *command line*, *tan display*, and *tangram display* are used to provide a reference in the discussion only.

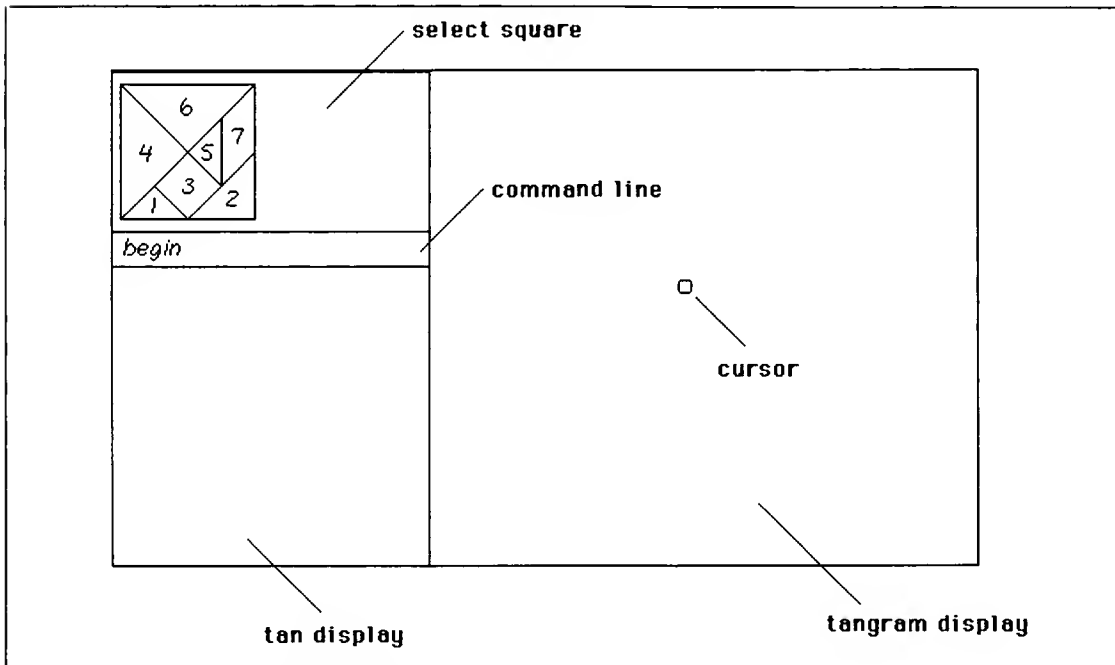


Illustration 12.1 The four screen output areas

The procedure for using this application is best described by briefly listing the operations used to produce a familiar tangram, and by displaying the screen image of each different step.

1. Upon typing "RUN", the program displays the four areas. The *select square* area shows the square tangram formed by the seven tans, each labeled by a number. The *command line* area shows, in Los Angeles font, the word "begin". The other two areas are empty (Illustration 12.2). The purpose of this display is to give you time to become familiar with the four areas.

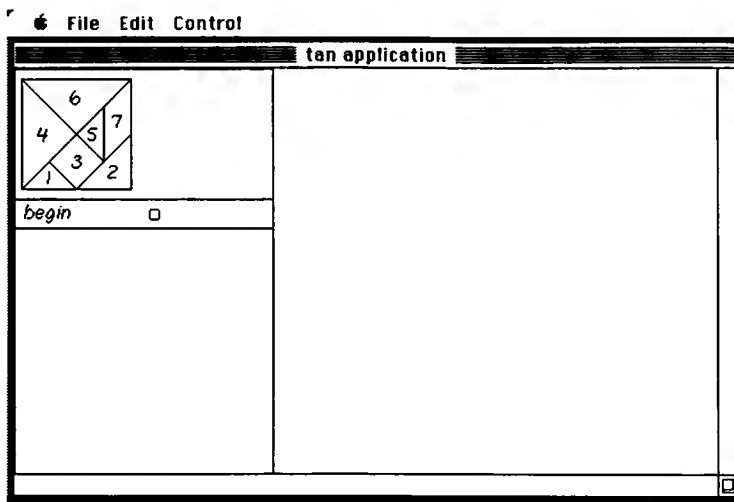


Illustration 12.2 Screen display after the command "RUN"

2. Position the cursor in the *command line* area and click the mouse. The *command line* changes from "begin" to "select tan" (Illustration 12.3).
3. Move the cursor over any one of the digits identifying the seven tans, and click. The selected tan is copied into the *tan display* area and begins to rotate in 45-degree increments, and the *command line* changes from "select tan" to "orient tan".
4. Move the cursor in the command line. When the tan that is rotating assumes the position you wish it to have in your tangram, click the mouse. The tan stops rotating and the command line changes from "orient tan" to "grab tan".
5. Move the cursor to the *tan display* area, to any one of the vertices. The cursor's hotspot must be within three pixels of the tan's vertex in order to be activated. When you click the mouse to "grab" the tan and the hotspot is close enough, an image of the tan vibrates, much as it would had you lassoed it in MacPaint. However, you do *not* have to drag the tan to the tangram display

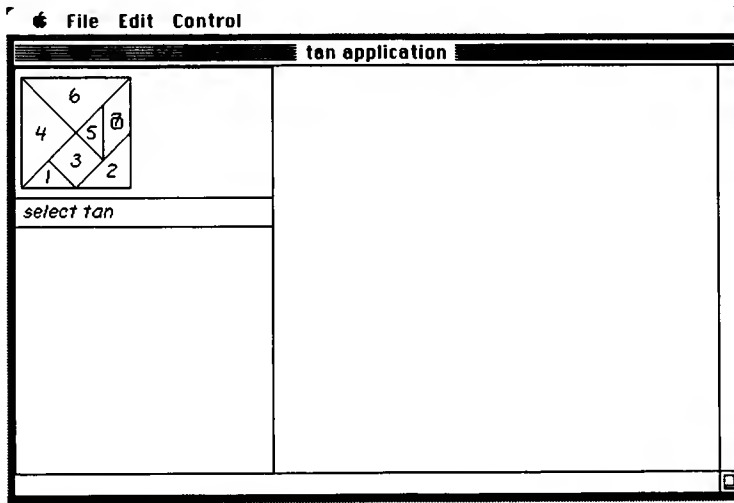


Illustration 12.3 Screen during tan selection

area. Moving the mouse is sufficient to move the tan (Illustration 12.4). Our thought was to simplify the move of the tan for you. You want to be careful about its placement, and you shouldn't have to worry about the mouse button being depressed during this careful operation.

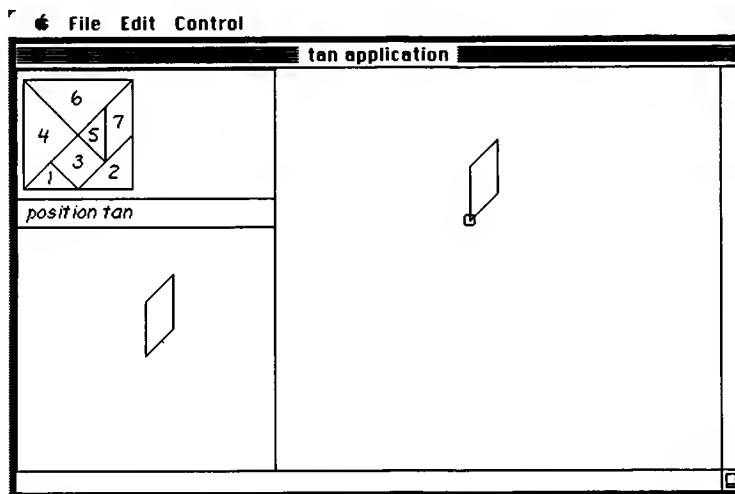


Illustration 12.4 Screen with tan moved to display area

6. To “let go” of the tan when you have it exactly where you want it. Click the mouse. The command line changes from “position tan” to “do undo”. This choice allows you to change your mind. For example, you might click the mouse at the wrong time during its rotation, so that its orientation is wrong. Simply click “undo” here and you go back to the *select tan* stage of the program. If you misplaced the tan, even though it was properly oriented, you can “undo” your mistake. In all cases, clicking the undo command will take you back to select the same or another tan.
7. As you repeat the steps 2 through 6 above, you develop your picture. When all seven tans have been selected, oriented, grabbed, and positioned, your tangram is done and the command line shows “tangram complete”. The three areas to the left of the tangram display area are erased, as shown in Illustration 12.5.

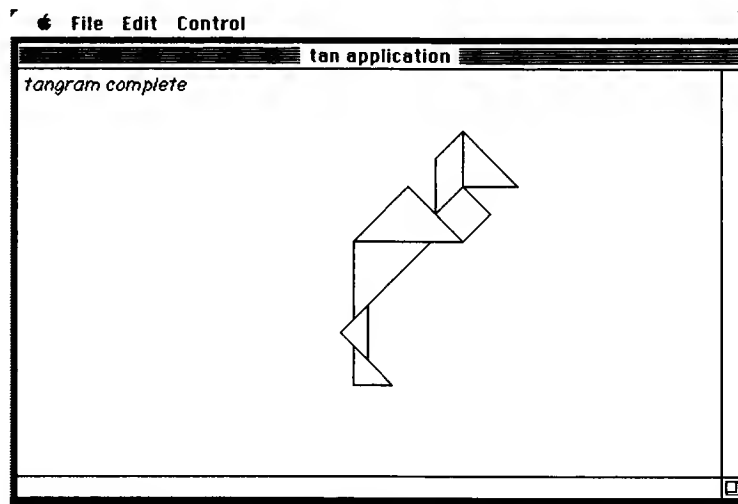


Illustration 12.5 Screen with completed stork tangram

The listing of the Tangrams program has several noteworthy features, among them a reliance on structure with many subroutines, and some heavy use of the Macintosh's Quickdraw ROM routines.

Listing, Tangrams program

```

10 ' tan application
20 DEFINT A-Z
30 DIM C(34),GP(233)
40 DIM TT(7),OT(7),VT(7),XT(7),YT(7)
50 DIM ST(4,3,2),MT(4,3,2),SQ(2,4,2),PA(4,4,2)
60 M1=7:M2=104:F=0
70 FOR I=0 TO 3
80 C(I)=0:C(15-I)=0
90 NEXT I
100 C(4)=&H7E0:C(11)=&H7E0
110 FOR I=5 TO 10
120 C(I)=&H810
130 NEXT I
140 FOR I=0 TO 15
150 C(I+16)=C(I)
160 NEXT I
170 C(32)=8:C(33)=8
180 CALL SETCURSOR(VARPTR(C(0)))
190 CLS
200 GOSUB 4000 ' draw tangram tile
210 FOR I=1 TO 4
220 FOR J= 1 TO 3
230 FOR K=1 TO 2
240 READ ST(I,J,K)
250 NEXT K
260 NEXT J
270 NEXT I
280 ' small triangles -tans 1 & 5
290 ' large triangles -tans 4 & 6 are 2x
300 DATA 0,0,19,-19,38,0
310 DATA 0,0,27,0,27,27
320 DATA 0,0,19,19,0,38
330 DATA 0,0,0,27,-27,27
340 FOR I=1 TO 4
350 FOR J=1 TO 3
360 FOR K=1 TO 2
370 READ MT(I,J,K)
380 NEXT K
390 NEXT J
400 NEXT I
410 ' medium triangle -tan 2
420 DATA 0,0,38,-38,38,0
430 DATA 0,0,54,0,27,27
440 DATA 0,0,38,38,0,38
450 DATA 0,0,0,54,-27,27
460 FOR I=1 TO 2
470 FOR J=1 TO 4

```

(continued)

```
480 FOR K=1 TO 2
490 READ SQ(I,J,K)
500 NEXT K
510 NEXT J
520 NEXT I
530 ' square -tan 3
540 DATA 0,0,19,-19,38,0,19,19
550 DATA 0,0,27,0,27,27,0,27
560 FOR I=1 TO 4
570 FOR J=1 TO 4
580 FOR K=1 TO 2
590 READ PA(I,J,K)
600 NEXT K
610 NEXT J
620 NEXT I
630 ' parallelogram -tan 7
640 DATA 0,0,0,-38,19,-57,19,-19
650 DATA 0,0,27,-27,54,-27,27,0
660 DATA 0,0,38,0,57,19,19,19
670 DATA 0,0,27,27,27,54,0,27
680 GOSUB 5000:PRINT"begin"
690 IF MOUSE(0)=1 THEN 700 ELSE 690
700 A=MOUSE(1):B=MOUSE(2)
710 IF B>90 AND B<110 AND A<180 THEN 730
720 GOTO 690
730 GOSUB 5000
740 IF F<7 THEN PRINT "select tan":GOTO 770 ELSE
      GOSUB 8000
750 CALL MOVETO(5,15):PRINT "tangram complete":CALL
      INITCURSOR
760 IF INKEY$="" THEN 760 ELSE STOP
770 LINE(1,111)-(179,299),30,BF
780 X=5:Y=7:T=0
790 IF (A>X+2 AND A<X+20) AND (B>Y+34 AND B<Y+52)
      THEN T=4
800 IF (A>X+6 AND A<X+24) AND (B>Y+64 AND B<Y+82)
      THEN T=1
810 IF (A>X+25 AND A<X+43) AND (B>Y+9 AND B<Y+27)
      THEN T=6
820 IF (A>X+26 AND A<X+44) AND (B>Y+53 AND B<Y+71)
      THEN T=3
830 IF (A>X+37 AND A<X+55) AND (B>Y+34 AND B<Y+52)
      THEN T=5
840 IF (A>X+54 AND A<X+72) AND (B>Y+25 AND B<Y+43)
      THEN T=7
850 IF (A>X+51 AND A<X+69) AND (B>Y+60 AND B<Y+78)
      THEN T=2
860 A=MOUSE(1):B=MOUSE(2)
870 IF MOUSE(0)=0 OR T=0 THEN 790
880 X=90:Y=200
```

```

890 O=1
900 GOSUB 2000 ' set orientation
910 GOSUB 3000 ' draw tan
920 GOSUB 5000:PRINT "orient tan"
930 A=MOUSE(1):B=MOUSE(2)
940 FOR Z=1 TO 1000:NEXT Z
950 IF MOUSE(0)=1 AND B>90 AND B<110 AND A<180 THEN
    1050
960 O=O+1
970 LINE(1,111)-(179,299),30,BF
980 IF T=7 AND O>16 THEN O=1
990 IF T=3 AND O>2 THEN O=1
1000 IF T<>3 AND T<>7 AND O>8 THEN O=1
1010 GOSUB 2000 ' set orientation
1020 GOSUB 3000 ' draw tan
1030 IF MOUSE(0)=1 THEN 1050
1040 GOTO 930
1050 GOSUB 5000:PRINT "grab tan"
1060 IF T=3 OR T=7 THEN L=4 ELSE L=3
1070 IF MOUSE(0)=1 THEN A=MOUSE(1):B=MOUSE(2):GOTO
    1090
1080 GOTO 1070
1090 FOR I=1 TO L
1100 IF ABS(A-P(1,1)-X)<3 AND ABS(B-P(1,2)-Y)<3
    THEN 1140
1110 A=MOUSE(1):B=MOUSE(2)
1120 NEXT I
1130 GOTO 1070
1140 XG=0:YG=0
1150 FOR I=1 TO L
1160 IF ABS(P(1,1))>ABS(XG) THEN XG=P(1,1)
1170 IF ABS(P(1,2))>ABS(YG) THEN YG=P(1,2)
1180 NEXT I
1190 GET(X,Y)-(X+XG,Y+YG),GP
1200 IF T=3 AND O=1 THEN GET(X,Y+YG)-(X+XG,Y-YG),GP
1210 GOSUB 5000:PRINT"position tan"
1220 X=MOUSE(1):Y=MOUSE(2)
1230 IF T=3 AND O=1 THEN 1270
1240 PUT(X,Y)-(X+XG,Y+YG),GP,XOR
1250 PUT(X,Y)-(X+XG,Y+YG),GP,XOR
1260 GOTO 1290
1270 PUT(X,Y+YG)-(X+XG,Y-YG),GP,XOR
1280 PUT(X,Y+YG)-(X+XG,Y-YG),GP,XOR
1290 IF MOUSE(0)=1 THEN 1300 ELSE 1220
1300 IF X<180 THEN 1210
1310 GOSUB 3000
1320 GOSUB 5000:PRINT"do"
1330 LINE(90,91)-(90,109):CALL MOVETO(90+M1,M2):
    PRINT"undo"

```

(continued)


```
1340 IF MOUSE(0)<>1 THEN 1330
1350 A=MOUSE(1):B=MOUSE(2)
1360 IF B>90 AND B<110 AND A>90 AND A<180 THEN
      GOSUB 7000 ELSE GOSUB 6000
1370 GOTO 730
1380 CALL TEXTMODE(0)
2000 ' set orientation
2010 IF T=3 THEN 2160
2020 IF T=7 THEN 2220
2030 IF T=2 THEN 2340
2040 IF T=1 THEN B=0+0:M=1
2050 IF T=5 THEN B=0+6:M=1
2060 IF T=4 THEN B=0+2:M=2
2070 IF T=6 THEN B=0+4:M=2
2080 IF B>8 THEN B=B-8
2090 IF B>4 THEN K=B-4:M=-M ELSE K=B
2100 FOR I=1 TO 3
2110 FOR J=1 TO 2
2120 P(I,J)=ST(K,I,J)*M
2130 NEXT J
2140 NEXT I
2150 RETURN
2160 FOR I=1 TO 4
2170 FOR J=1 TO 2
2180 P(I,J)=SQ(0,I,J)
2190 NEXT J
2200 NEXT I
2210 RETURN
2220 B=(0-1) MOD 8 + 1
2230 IF B>4 THEN K=B-4:M=-1 ELSE K=B:M=1
2240 FOR I=1 TO 4
2250 FOR J=1 TO 2
2260 P(I,J)=PA(K,I,J)*M
2270 NEXT J
2280 NEXT I
2290 IF 0<9 THEN RETURN
2300 FOR I=1 TO 4
2310 P(I,1)=-P(I,1)
2320 NEXT I
2330 RETURN
2340 IF 0>4 THEN K=0-4:M=-1 ELSE K=0:M=1
2350 FOR I=1 TO 3
2360 FOR J=1 TO 2
2370 P(I,J)=MT(K,I,J)*M
2380 NEXT J
2390 NEXT I
2400 RETURN
3000 ' draw tan
3010 IF T=3 OR T=7 THEN L=3 ELSE L=2
3020 FOR I=1 TO L
```

```

3030 LINE(X+P(I,1),Y+P(I,2))-(X+P(I+1,1),Y+P
      (I+1,2))
3040 NEXT I
3050 LINE(X+P(L+1,1),Y+P(L+1,2))-(X+P(1,1),Y+P
      (1,2))
3060 RETURN
4000 ' draw tangram tile
4010 X=0:Y=0
4020 LINE(0,0)-(180,300),,B
4030 LINE(0,90)-(180,110),,B
4040 X=X+5:Y=Y+7
4050 LINE(X,Y)-(X+76,Y+76),,B
4060 LINE(X,Y+76)-(X+76,Y)
4070 LINE(X+38,Y+76)-(X+76,Y+38)
4080 LINE(X,Y)-(X+57,Y+57)
4090 LINE(X+57,Y+57)-(X+57,Y+19)
4100 LINE(X+19,Y+57)-(X+38,Y+76)
4110 CALL TEXTMODE(1):CALL TEXTFONT(12)
4120 CALL MOVETO(X+15,Y+73):PRINT "1"
4130 CALL MOVETO(X+60,Y+69):PRINT "2"
4140 CALL MOVETO(X+35,Y+62):PRINT "3"
4150 CALL MOVETO(X+11,Y+43):PRINT "4"
4160 CALL MOVETO(X+46,Y+43):PRINT "5"
4170 CALL MOVETO(X+34,Y+18):PRINT "6"
4180 CALL MOVETO(X+63,Y+34):PRINT "7"
4190 CALL TEXTMODE(0)
4200 RETURN
5000 ' prompt manager
5010 LINE(1,91)-(179,109),30,BF
5020 CALL MOVETO(M1,M2)
5030 CALL TEXTMODE(1)
5040 RETURN
6000 ' save tan info
6010 F=F+1
6020 TT(F)=T:OT(F)=O:XT(F)=X:YT(F)=Y
6030 RETURN
7000 ' undo
7010 LINE(181,1)-(499,299),30,BF
7020 FOR Z=1 TO F
7030 T=TT(Z):O=OT(Z):X=XT(Z):Y=YT(Z)
7040 GOSUB 2000:GOSUB 3000
7050 NEXT Z
7060 RETURN
8000 ' wrap up
8010 GOSUB 5000:PRINT "tangram complete"
8020 LINE (0,0)-(500,300),30,BF
8030 XD=100:YD=100
8040 FOR Z2=1 TO F
8050 T=TT(Z2):O=OT(Z2):X=XT(Z2):Y=YT(Z2)

```

(continued)

```

8060 GOSUB 2000
8070 IF Z2=P1 THEN XS=X-XD:YS=Y-YD
8080 X=X-XS:Y=Y-YS
8090 GOSUB 3000
8100 NEXT Z2
8110 RETURN

```

Commonly Used Variables

Let's review some of the most commonly used variables.

C — new cursor. "C" is an integer array DIMensioned 34. Cursors in BASIC are most easily defined in hex images of a binary 16-bit number. The cursor design begins as a 16x16 grid, with the filled (dark) pixels marked and the "Hot Spot" pixel pair identified. Illustration 12.6 shows the design for our cursor shaped like a rounded square with the center dot being the "Hot Spot".

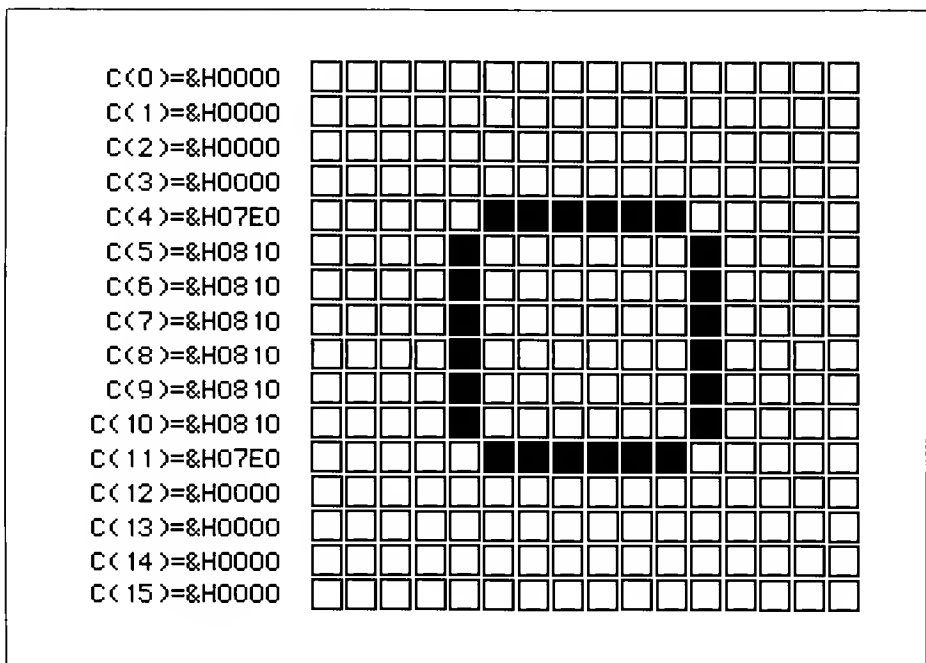


Illustration 12.6 Square cursor in 16x16 grid

Each grid position is a bit, and each row in the grid is a 16-bit integer. In binary, for example, the fifth row is "0000011111100000", which is the hex number "7E0". To define the cursor in BASIC, you define the first 16 positions of the cursor array, C(0) through C(15), as follows:

1. Define first four rows as hex zeros
2. Define the fifth and twelfth rows as hex 7E0
3. Define rows six through 11 as hex 810

The cursor's "Hot Spot" is defined as X-Y coordinates in C(32) and C(33). It isn't a single pixel. Rather, it is the intersecting point between four pixels, the one identified by coordinates, the one to its left, the one above, and the one above and to the left (Illustration 12.7).

In our cursor, the "Hot Spot" is defined as

C(32)=8: C(33)=8 'vertical and horizontal coordinates

The BASIC command **CALL INITCURSOR**: restores the system cursor, and the command.

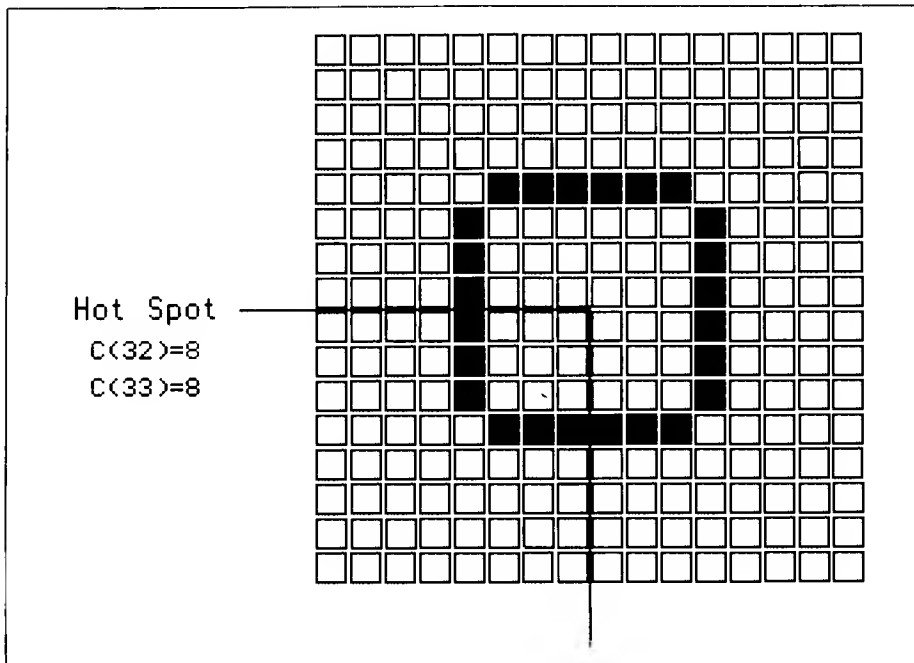


Illustration 12.7 Hot spot intersection

CALL SETCURSOR(VARPTR(C(0))): sets our cursor to be the active one.

While we're on the subject of ROM routines that deal with the cursor, we should mention some others that you may wish to use to embellish your applications.

CALL HIDECURSOR: turns off the cursor so that it becomes invisible, even if the mouse is dragged.

CALL SHOWCURSOR: is the reverse, making the mouse visible.

CALL OBSCURECURSOR: is an interesting variant of HIDECURSOR.

It hides the mouse cursor until the mouse moves. In the MacWrite program, for example, the cursor disappears when a character is typed, and appears only when the mouse is moved. This is especially nice when you want a clean screen with the cursor out of the way until you need it.

Other variables

GP — contains a picture of the tan piece that was selected, oriented, and positioned. The largest tan is Tan #4 or Tan #6. The picture is stored as an integer array for GETs and PUTs. Illustration 12.8 indicates the space needed in GP to store the tans.

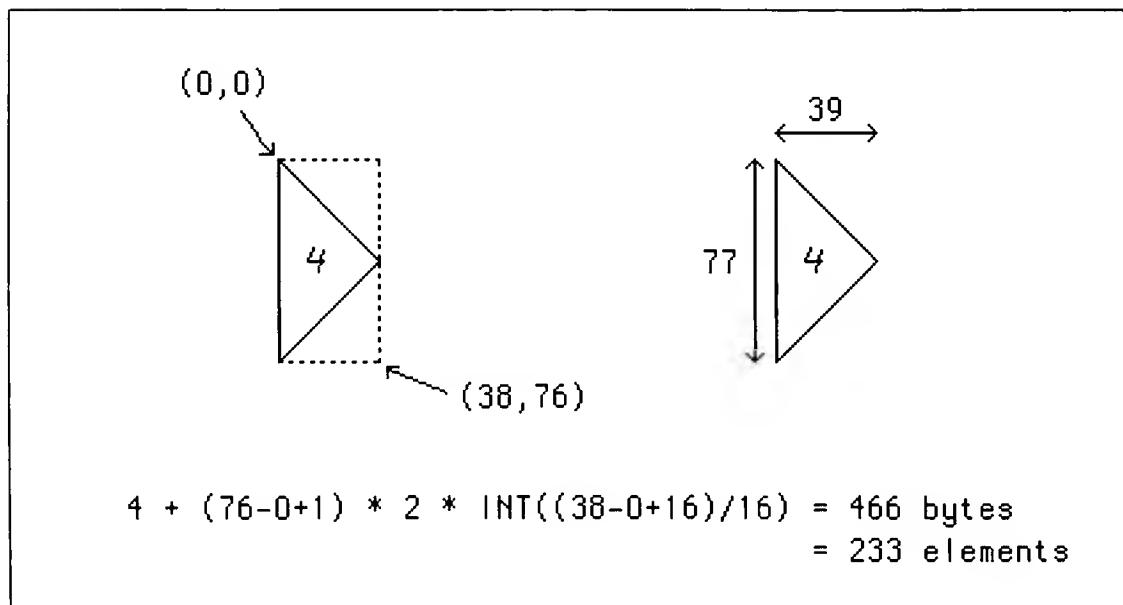


Illustration 12.8 Large triangle with corners marked

The array must be DIMensioned to hold 233 integers, according to the formula from the Microsoft BASIC manual:

$$4 + (76 - 0 + 1) * 2 * \text{INT}((38 - 0 + 16) / 16) \\ = 466 \text{ bytes, or } 233 \text{ 16-bit integers}$$

TT — Contains the tan piece numbers. TT(1) might be 3 if the square were the first tan piece manipulated.

OT — orientation of the tan piece. For tan pieces 1, 2, 4, 5, 6 there are eight orientations for each of the pieces: $45^\circ \times 8 = 360^\circ$.

XT, YT — absolute coordinates of final position for tan TT.

ST — Small triangles (tans #1 and #5) and large triangles which are twice the size of the small (tans #4 and #6). Because of symmetry, only four orientations need to be stored, as all others are derived from them. See the Set Orientation subroutine at lines 1400-1800.

P — relative coordinates of tan's vertices after orientation has been selected.

MT — Medium triangle (tan #2). The same comments apply here as do above for the small triangles.

SQ — square (tan #3). Only has two orientations, but four vertices. $2 \times 4 \times 2$.

PA — parallelogram (tan #7). Has eight orientations, but has eight more when flipped. Still, only four orientations need to be stored, because the rest of them can be derived from those four.

M1, M2 — X and Y coordinates of the beginning of the command line area. F is the count of tans used. NOTE: Although the number of tans is checked, there is no check on whether a specific tan is used more than once. Thus, a tangram with seven squares, or seven parallelograms.

70-180	Define a circular cursor with hot spot at center.
200	Subroutine to draw initial screen with tangram tile menu in the <i>select tan</i> area.
210-670	Load tan arrays with orientation information.
680	Prompt Manager at 5000 to position the cursor for printing in the <i>command line</i> area.
690	If mouse button not clicked, cycle.
700-710	Check to see if the <i>command line</i> area has been clicked. If so, store coordinates in A and B. If not, GOSUB 5000 to position cursor at prompt area.
740-760	As long as more tans are available ($F < 7$), "select tan" and continue, else we're done. Print "tangram complete" and wait for keystroke.
770	Clear the <i>select tan</i> area.
780	This is X,Y position of upper left corner of <i>select tan</i> area.
790-850	IF filter to see which one of the seven tans is being selected (single-clicked).
880	This X,Y will be the position of the selected tan when drawn in the <i>tan orientation</i> area.

890	We begin with orientation $0 = 1$ and step through orientations at 45° increments until a click occurs with mouse pointing in prompt area. Each tan selected starts at its original orientation in the square tangram tile.
900	GOSUB 3000 to set orientation. Actually defines the vertices of the selected tan in array P(i,j)
910-930	Redraws tan with new orientation, prompts user.
940	Pause.
950-960	If mouse clicked in prompt area, orientation has been selected, else show its next orientation.
970	Clear orientation area.
980-1000	Keep track of wraparound counting of angle.
1050	Prepare to grab tan piece whose orientation has been selected.
1060	Number of vertices.
1070-1130	Need to click on a vertex to "grab" the tan.
1140-1180	Need to find perimeter of tan piece (the smallest rectangle into which it will fit) in preparation for a GET at line 1200.
1200	Tan #3, the square, is a special case if orientation is 1.
1210	Prepare to position tan piece in positioning area.
1220-1300	Grab the tan. As mouse moves, tan will follow it. The two PUTs at 1240-1250 or 1270-1280 (tan #3 is still a special case) XOR it along without changing the background.
1320-1360	Click the "do" in the prompt area and the tan is in position forever. Click the "undo" and the tan is erased with any previously positioned tans redrawn.
1370	Get another tan piece?
2000	Set orientation subroutine. Manipulations to establish the relative vertices of the tan piece selected and oriented.
3000	Draw the tan. Given coordinates X,Y of where the tan is to go, it is simply a matter of three or four lines being drawn.
3000	Draw tangram tile. Sets up screen with tangram tile menu and number in place. Los Angeles type font selected with TEXTMODE (1). Printing on top of background without destroying it.
5000	Prompt manager.
6000	Save tan information. When a tan is finally positioned, we need to remember which tan T, what orientation Theta, where the tan T is with orientation O was drawn (X,Y)
7000	Redo (undo) routine. Clear entire tan orientation area. The picture being drawn is erased! Not to worry, however. We've remembered all important information about the previously positioned tans. Redraws the tans already placed.
8000	Wrap-up. Clear screen and redraw tangram developed by user. Perhaps this routine could display the saved tan information for later redrawing of the figure. That's one of the many embellishments we leave to you.

Consider the possibilities! Why tans? Why not some other design primitives that you can describe to the program by way of the same arrangement of menu, command, orientation, and sketch areas? What about a scaling routine? Think how easy it would be to design a house plan with design primitives such as doors, walls, windows, closets, appliances, what have you. Or what about a flowchart, or organization chart, or HIPO chart, or PERT chart drawing program?

This is what the Tangrams application is all about — not the simplistic design of tangrams, although that is a great deal of fun — but the translation of this program into another application tuned to a specific area or profession. With some imagination, and with this program as a skeletal start, you should be capable of developing a truly strong application worthy of the marketplace.

CHART APPLICATIONS

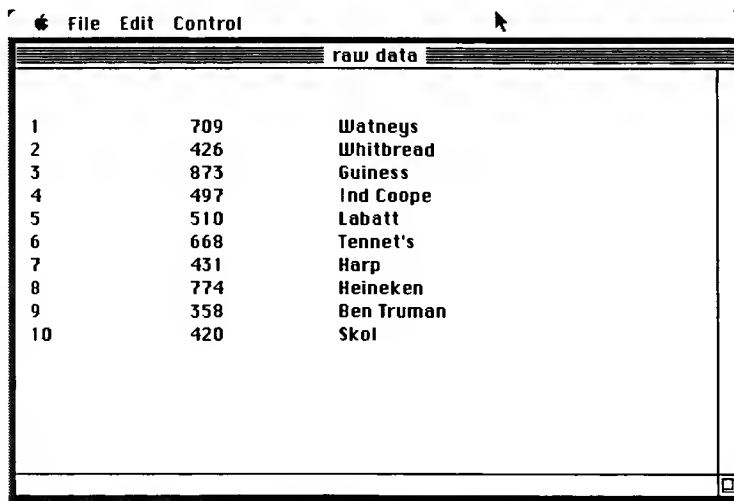
In this chapter, we will discuss their features in some detail, especially their use of the Macintosh Quickdraw routines.

197

business graphics applications. The idea is that the data will consist of both numbers and titles. Therefore our graphing programs must handle the numeric data indicating the proportion of each component, and they must provide string labels for each component.

The program includes the names of ten foreign breweries and their associated annual production for a given hypothetical year, say in zillions of barrels. The names were chosen to be typical of string data. Tennet's has an apostrophe; Ben Truman is two names; and the lengths of the names vary from Skol to Ben Truman. The numeric data also varies, from 426 to 873, a typical range of values in business charts.

In order to test our chart-making applications thoroughly, we designed this program to provide any number of breweries' names and production, from two to all ten. Thus the programs must be able to display just two breweries, or deal with up to ten breweries; and each display must be appropriately scaled and centered.



1	709	Watneys
2	426	Whitbread
3	873	Guinness
4	497	Ind Coope
5	510	Labatt
6	668	Tennet's
7	431	Harp
8	774	Heineken
9	358	Ben Truman
10	420	Skol

Illustration 13.1 Typical run of Raw Data program

Listing, RAWDATA program

```

10 ' raw data
20 DIM X(10),T$(10)
30 CLS
40 RANDOMIZE TIMER
50 CALL TEXTFONT(0)
60 READ N
70 FOR I=1 TO N
80 READ X(I),T$(I)
90 NEXT I
100 DATA 10,709,"Watneys"

```

```

110 DATA 426,"Whitbread",873,"Guinness",497,"Ind
    Coope"
120 DATA 510,"Labatt",668,"Tennet's",431,"Harp"
130 DATA 774,"Heineken",358,"Ben Truman",420,"Skol"
140 N=2+INT((N-1)*RND)
150 PRINT:PRINT
160 FOR I=1 TO N
170 PRINT I,X(I),T$(I)
180 NEXT I
190 IF MOUSE(0)=1 THEN RESTORE: GOTO 30 ELSE 190

```

20	Define 10 numeric values X, 10 string titles T\$
30	Clear the screen
40	Seed random number generator with random seed
50	Set textfont to Chicago
60	Read N, number of breweries
70-90	Read numeric production and brewery names into X and T\$.
100-130	Establish all data in DATA statements
140	Define new N as random integer from 2 to 10
150-180	Print selected breweries and their production
190	Sense mouse. If clicked, restore and repeat process

Notice that the program will always display Watneys and Whitbread, regardless of the number of breweries selected, because they are the first two in the list. In order to select two to 10 breweries at random from the pool in the DATA statements, you can modify the program. Alter the program to shuffle the X and T\$ arrays before selecting the random number N and printing the random breweries.

```

92 FOR I=1 TO 20
94   P1=INT(RND*10+1): P2=INT(RND*10+1)
96   SWAP T$(P1),T$(P2): SWAP X(P1),X(P2)
98 NEXT I

```

The four lines above effectively shuffle the ten brewery names and numeric values.

Application 1: Piechart

Before we list and discuss the program and its output, let us describe how you can establish several patterns for use in any program. Remember that any screen image is produced by displaying integers in the Macintosh memory. Many of the Quickdraw routines allow you to paint areas with patterns you have defined, instead of default system pattern. When you define a pattern to the Macintosh, you must define an array of four integers, each of which is stored as a 16-bit binary

value in memory. These 64 bits are rearranged internally to become an 8-by-8-bit grid, each signifying a pixel.

For example, suppose you want a pattern that looks like little squares, as in Illustration 13.2.

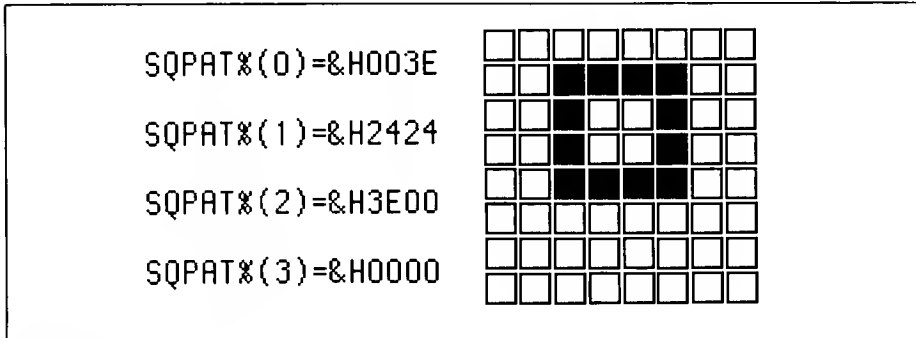


Illustration 13.2 Definition of pattern

You define the four 16-bit integers as follows:

```

20 DIM SQPAT$PC(3) 'really four integers, SQPAT$PC(0) through SQPAT$PC(3)
30 SQPAT$PC(0) = &H003E 'binary 0000000000111100, first 2 rows
40 SQPAT$PC(1) = &H2424 'binary 0010010000100100, second 2 rows
50 SQPAT$PC(2) = &H3E00 'binary 0011110000000000, third 2 rows
60 SQPAT$PC(3) = &H0000 'binary 0000000000000000, fourth 2 rows
  
```

You can define more than one pattern in the same array by using a second subscript as a pattern number. We have done this in these programs, and we recommend it to you as a common practice when you have more than one pattern to deal with. Here's how you do it:

1. Define an integer array (let's call it P%) DIMensioned 3 by the number of patterns you want (let's say 10).
2. Place first pattern in P%(0,1), P%(1,1), P%(2,1), and P%(3,1).
3. Place second pattern in P%(0,2), P%(1,2), P%(2,2), and P%(3,2).
4. Continue until all ten patterns are defined.

Hint: We recommend that you establish two patterns, grey and black, as a matter of course whenever you work with multiple patterns.

Black is &HFFFF, &HFFFF, &HFFFF, &HFFFF where you define all dots in the 8-by-8 grid as 1s. Another way to do this is to define the four pattern integers as -1, because the integer -1 is a binary 11111111111111, which is a hex FFFF.

Grey is &H55AA, &H55AA, &H55AA, &H55AA.

When you use patterns with a CALL to the Macintosh Quickdraw routines, you must use the VARPTR function. You will always refer to a pattern as, for example, VARPTR(SQPAT\$PC(0)) or VARPTR(P%(0,1)) or VARPTR(P%(0,2)). Illustration 13.3 and 13.4 are examples of pie charts using patterns. The two illustrations are followed by the piechart listing.

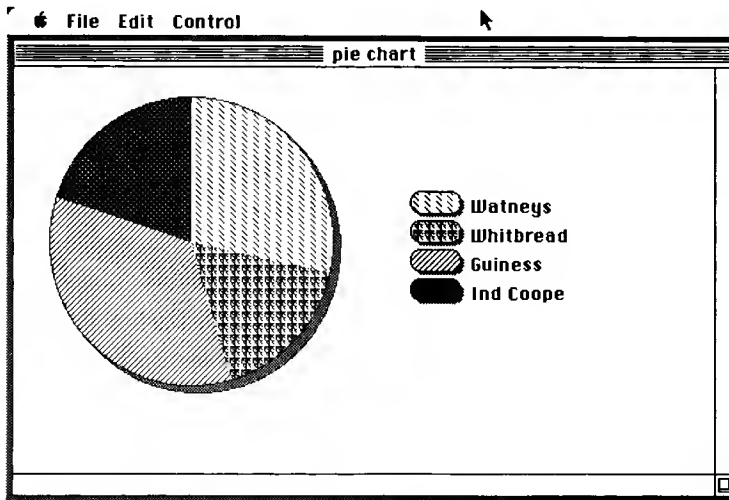


Illustration 13.3 Piechart of a few breweries

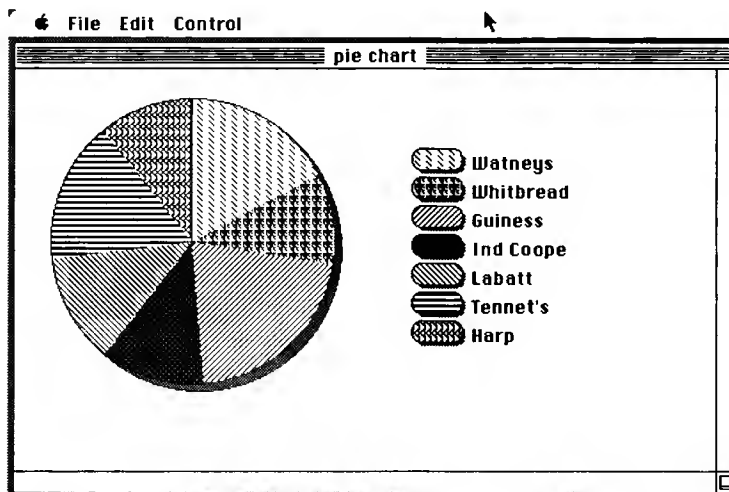


Illustration 13.4 Piechart of many breweries

Listing, Piechart

```
10 ' pie chart
20 DIM X(10), P%(3,12), R%(3), L%(3), T$(10)
30 CLS
40 RANOMIZE TIMER
50 CALL TEXTFONT(0)
60 READ N
70 FOR I=1 TO N
80 READ X(I), T$(I)
90 NEXT I
100 DATA 10,709,"Watneys"
110 DATA 426,"Whitbread",873,"Guinness",497,
    "Ind Coope"
120 DATA 510,"Labatt",668,"Tennet's",431,"Harp"
130 DATA 774,"Heineken",358,"Ben Truman",420,"Skol"
140 N=2+INT((N-1)*RND)
150 GOSUB 1000 ' <<<< normalize data
160 P%(0,1)=&H55AA:P%(1,1)=&H55AA:P%(2,1)=
    &H55AA:P%(3,1)=&H55AA
170 P%(0,2)=-1:P%(1,2)=-1:P%(2,2)=-1:P%(3,2)=-1
180 P%(0,3)=&H8040:P%(1,3)=&H2010:P%(2,3)=
    &H8040:P%(3,3)=&H2010
190 P%(0,4)=&HFEOC:P%(1,4)=&HBA98:P%(2,4)=
    &H7654:P%(3,4)=&H3210
200 P%(0,5)=&H1122:P%(1,5)=&H4488:P%(2,5)=
    &H1122:P%(3,5)=&H4488
210 P%(0,6)=&H77FF:P%(1,6)=&H00FF:P%(2,6)=
    &H77FF:P%(3,6)=&H00FF
220 P%(0,7)=&H8844:P%(1,7)=&H2211:P%(2,7)=
    &H8844:P%(3,7)=&H2211
230 P%(0,8)=-1:P%(1,8)=0:P%(2,8)=-1:P%(3,8)=0
240 P%(0,9)=&H9966:P%(1,9)=&H3311:P%(2,9)=
    &H1133:P%(3,9)=&H6699
250 P%(0,10)=&HCCCC:P%(1,10)=&H3333:P%(2,10)=
    &HCCCC:P%(3,10)=&H3333
260 P%(0,11)=&H33AA:P%(1,11)=&H33AA:P%(2,11)=
    &H33AA:P%(3,11)=&H33AA
270 P%(0,12)=&H1111:P%(1,12)=&H1111:P%(2,12)=
    &H1111:P%(3,12)=&H1111
280 R%(0)=25:R%(1)=30:R%(2)=225:R%(3)=230
290 CALL FILLOVAL(VARPTR(R%(0)),VARPTR(P%(0,1)))
300 CALL FRAMEOVAL(VARPTR(R%(0)))
310 FOR J=0 TO 3
320 R%(J)=R%(J)-5
330 NEXT J
340 CALL FILLOVAL(VARPTR(R%(0)),VARPTR(P%(0,2)))
350 A=0
360 S=10*(10-N)+7
370 E=280
```

```

380 FOR I=1 TO N
390 W=INT(X(I)*360+.5)
400 CALL FILLARC(VARPTR(R%(0)),A,W,VARPTR(P%(0,2)))
410 CALL FILLARC(VARPTR(R%(0)),A,W,VARPTR
    (P%(0,I+2)))
420 A=A+W
430 D=20*I+S
440 L%(0)=0:L%(1)=E:L%(2)=0+16:L%(3)=E+36
450 CALL FILLROUNDRECT(VARPTR(L%(0)),15,15,VARPTR
    (P%(0,1)))
460 FOR J=0 TO 3
470 L%(J)=L%(J)-2
480 NEXT J
490 CALL FILLROUNDRECT(VARPTR(L%(0)),15,15,VARPTR
    (P%(0,I+2)))
500 CALL FRAMEROUNDORECT(VARPTR(L%(0)),15,15)
510 CALL MOVETO(320,0+13):PRINT T$(I)
520 CALL FRAMEOVAL(VARPTR(R%(0)))
530 NEXT I
540 IF MOUSE(0)=1 THEN RESTORE: GOTO 30 ELSE 540
1000 ' data normalization
1010 S8=X(1)
1020 FOR I8=2 TO N
1030 S8=S8+X(I8)
1040 NEXT I8
1050 FOR I8=1 TO N
1060 X(I8)=X(I8)/S8
1070 NEXT I8
1080 RETURN

```

20	Define X and T\$ as production and brewery names. Define P% for 12 patterns. Define R% for rectangle coordinates. Define L% for filled rectangle coordinates.
30-140	Establish a random number N of breweries to be graphed.
150	Perform Normalize Data subroutine (lines 1000-1080)
160	Define grey pattern starting at P%(0,1).
170	Define black pattern starting at P%(0,2).
180-270	Define 10 patterns, one for each different brewery, at P%(0,3) through P%(0,12).
280	Define R% as rectangle with upper-left and lower-right coordinates of (15,10) and (215,210). It must hold a 200-pixel-diameter circle.
290-300	Fill oval with grey pattern, frame it.
310-340	Shift coordinates of oval, draw another.
350	Start angle accumulator A to zero.
360-370	Define S and E to position text on chart.
380-530	For I = 1 to N do:

390	Set size of wedge W.
400-410	Fill wedge with its pattern (# 1 + 2).
420	Accumulate angle A, now starting position for next wedge.
430	Calculate displacement D used for rounded rectangle containing appropriate pattern (# 1 + 2).
440	Define upper-left, lower-right coordinates for rounded rectangle as (D,E) and (D + 16,E + 36).
450	Fill rounded rectangle associating wedge and text.
460-500	Shift coordinates, draw again.
510-520	Print text next to this rectangle.
530	Enddo.
540	Sense mouse. If clicked, restore data and return to show another chart.
1000	Data normalization subroutine.
1010	Set X8 as first value in array X.
1020-1040	Calculate S8, sum of all N elements of X.
1050-1070	Calculate new X for each position dividing old X by S8. This makes all X values proportional, but none greater than 1.

Application 2: Icon Chart

This program produces a chart like those you've seen many times in magazines, in which the value of a variable is shown as a symbol of what it is (Illustration 13.5 and 13.6). For example, car production could be shown as different-sized symbols of cars, or the population of several countries could be shown as several symbolic people of varying size. These charts are sometimes called *ideograms*, but we prefer to call them *icon charts* to conform to the Macintosh's terminology.

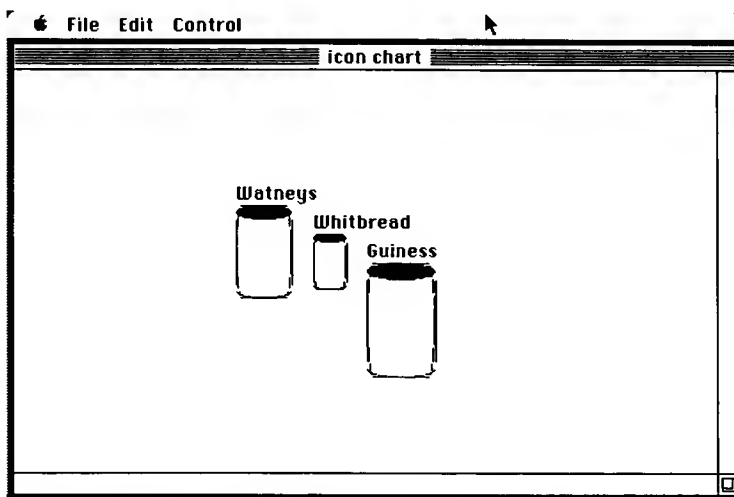


Illustration 13.5 Icon chart output with few breweries

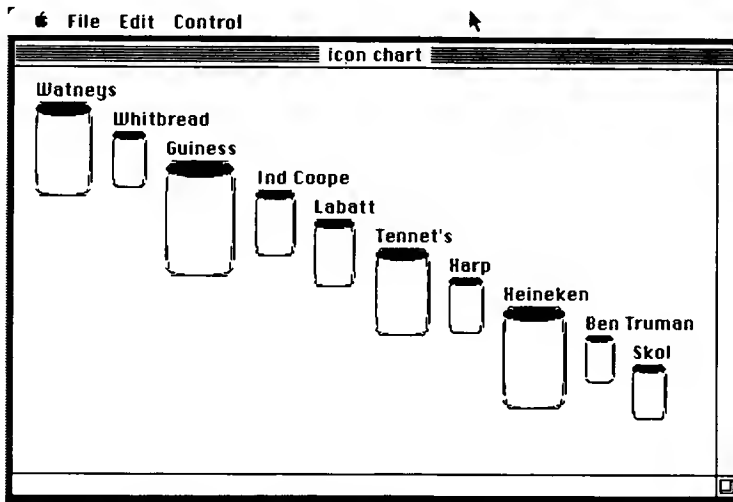


Illustration 13.6 Icon chart output with many breweries

Listing, Icon Chart

```

10 ' icon chart
20 DIM X(10),P%(3),R%(3),L%(3),T$(10)
30 CLS
40 RANOMIZE TIMER
50 CALL TEXTFONT(0)
60 READ N
70 FOR I=1 TO N
80 READ X(I),T$(I)
90 NEXT I
100 DATA 10,709,"Watneys"
110 DATA 426,"Whitbread",873,"Guinness",497,"Ind
    Coope"
120 DATA 510,"Labatt",668,"Tennet's",431,"Harp"
130 DATA 774,"Heineken",358,"Ben Truman",420,"Skol"
140 N=2+INT((N-1)*RND)
150 GOSUB 1000 ' <<<< normalize data
160 P%(0)=&H55AA:P%(1)=&H55AA:P%(2)=
    &H55AA:P%(3)=&H55AA
170 X=15+40*(10-N)/2
180 FOR I=1 TO N
190 S=80*X(I)
200 Y=20*I+10*(10-N)+4
210 GOSUB 2000
220 D=16*I-15
230 CALL MOVETO(X,Y-4):PRINT T$(I)
240 X=X+50*X(I)+13

```

(continued)

```
250 NEXT I
260 IF MOUSE(0)=1 THEN RESTORE: GOTO 30 ELSE 260
1000 ' data normalization
1010 S8=X(1)
1020 FOR I8=2 TO N
1030 IF S8<X(I8) THEN S8=X(I8)
1040 NEXT I8
1050 FOR I8=1 TO N
1060 X(I8)=X(I8)/S8
1070 NEXT I8
1080 RETURN
2000 ' draw icon
2010 P4%=S:P5%=.6*S
2020 P1%=.12*S:P2%=.85*S:P3%=.15*S
2030 R%(0)=Y:R%(1)=X:R%(2)=Y+P4%:R%(3)=X+P5%
2040 CALL FRAMEROUNDRECT(VARPTR(R%(0)),P1%,P4%)
2050 LINE(X,Y+P3%)-(X,Y+P2%):LINE(X+P5%,Y+P3%)-
      (X+P5%,Y+P2%)
2060 R%(0)=Y+P2%:R%(2)=Y+P4%
2070 CALL FRAMEARC(VARPTR(R%(0)),90,180)
2080 R%(0)=Y:R%(2)=Y+P3%
2090 CALL FILLOVAL(VARPTR(R%(0)),VARPTR(P%(0)))
2100 CALL FRAMEOVAL(VARPTR(R%(0)))
2110 RETURN
```

- 20-160 See discussion in previous program. All stays the same, except that only one pattern is defined, a grey, and it is stored in the array P% in line 160. Line 20 shows that P% is singly dimensioned, as we deal with only one pattern.
- 170 Define X as horizontal starting position, further over if N is small, because we want to center the icons.
- 180-250 For I = 1 to N DO:
- 190 Set S, scale of icon, to max. vertical size = 80.
- 200 Set Y, width of icon, to max. = 48.
- 210 Perform Draw Icon routine.
- 220 Locate position for text, print brewery name.
- 230 Adjust X so that next can is over 13 pixels + its proper proportional width.
- 240 Enddo.
- 250 Sense mouse. If clicked, return for another icon chart.
- 1000-1080 Data normalization routine. Notice that it is different from the last one. This one divides every value in X by the largest value in X.
- 2000 Draw Icon Routine
- 2010-2020 Define P1% through P6% as dimensions based on scale S.
P1% = .12S, is oval width for can's bottom.

```

P2 % = .85S, is height of edge of beer can.
P3 % = .15S, is width of can's top.
P4 % = S, is oval height for can's bottom.
P5 % = .6S, is can's width to height ratio.
2030 Establish round rectangle's coordinates based on X, Y, P4 %,
P5 %.
2040 Draw can's outline using FRAMEROUNDRECT.
2060 Outline can with straight lines as well.
2070 Set up bottom of can's arc using P2 % and P4 %.
2080 Draw can's bottom using FRAMEARC.
2090 Set up can's top for shading using P3 %.
2100 Shade can's top grey using pattern P % and FILLOVAL.
2110 Draw can's top edge using FRAMEOVAL.
2120 return

```

Application 3: Bar Chart

The bar chart remains one of the most popular graphic displays for showing relative differences among several variables. It suits our example of the European breweries well, for that is what we have — from two to 10 numeric values whose differences are hard to detect. It is also difficult to get a sense of who's big and who's small in any list of numbers if that list has more than three or four elements. The bar chart makes this easy by making the larger values stand out above the crowd, and the smaller values look small because of their short bars (Illustration 13.7 and 13.8).

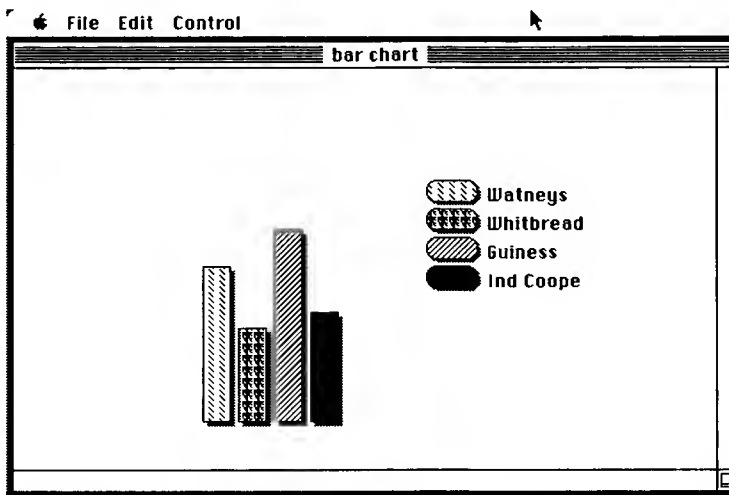


Illustration 13.7 Bar chart, few breweries

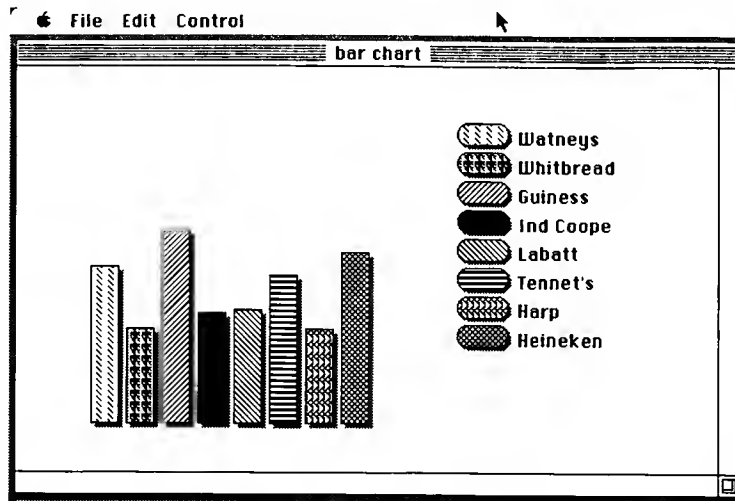


Illustration 13.8 Bar chart output, many breweries

Listing, Bar Chart Program

```

10 ' bar chart
20 DIM X(10),P%(3,12),R%(3),L%(3),T$(10)
30 CLS
40 RANOOIMIZE TIMER
50 CALL TEXTFONT(0)
60 READ N
70 FOR I=1 TO N
80 READ X(I),T$(I)
90 NEXT I
100 DATA 10,709,"Watneys"
110 DATA 426,"Whitbread",873,"Guinness",497,
    "Ind Coope"
120 DATA 510,"Labatt",668,"Tennet's",431,"Harp"
130 DATA 774,"Heineken",358,"Ben Truman",420,"Skol"
140 N=2+INT((N-1)*RND)
150 GOSUB 1000 ' <<<< normalize data
160 P%(0,1)=&H55AA:P%(1,1)=&H55AA:P%(2,1)=&H55AA:P%
    (3,1)=&H55AA
170 P%(0,2)=-1:P%(1,2)=-1:P%(2,2)=-1:P%(3,2)=-1
180 P%(0,3)=&H8040:P%(1,3)=&H2010:P%(2,3)=
    &H8040:P%(3,3)=&H2010
190 P%(0,4)=&HFEDC:P%(1,4)=&HBA98:P%(2,4)=
    &H7654:P%(3,4)=&H3210
200 P%(0,5)=&H1122:P%(1,5)=&H4488:P%(2,5)=
    &H1122:P%(3,5)=&H4488
210 P%(0,6)=&H77FF:P%(1,6)=&H00FF:P%(2,6)=
    &H77FF:P%(3,6)=&H00FF

```

```

220 P%(0,7)=&H8844:P%(1,7)=&H2211:P%(2,7)=
    &H8844:P%(3,7)=&H2211
230 P%(0,8)=-1:P%(1,8)=0:P%(2,8)=-1:P%(3,8)=0
240 P%(0,9)=&H9966:P%(1,9)=&H3311:P%(2,9)=
    &H1133:P%(3,9)=&H6699
250 P%(0,10)=&HCCCC:P%(1,10)=&H3333:P%(2,10)=
    &HCCCC:P%(3,10)=&H3333
260 P%(0,11)=&H33AA:P%(1,11)=&H33AA:P%(2,11)=
    &H33AA:P%(3,11)=&H33AA
270 P%(0,12)=&H1111:P%(1,12)=&H1111:P%(2,12)=
    &H1111:P%(3,12)=&H1111
280 W=20:H=100
290 S=10*(10-N)
300 X=15+S+S
310 E=X+55+25*N:B=250
320 FOR I=1 TO N
330 V=X(I)*H
340 R%(0)=B-V:R%(1)=X:R%(2)=B:R%(3)=X+W
350 CALL FILLRECT(VARPTR(R%(0)),VARPTR(P%(0,1)))
360 FOR J=0 TO 3
370 R%(J)=R%(J)-3
380 NEXT J
390 CALL FILLRECT(VARPTR(R%(0)),VARPTR(P%(0,I+2)))
400 CALL FRAMERECT(VARPTR(R%(0)))
410 X=X+25
420 D=20*I+S
430 L%(0)=D:L%(1)=E:L%(2)=D+16:L%(3)=E+36
440 CALL FILLROUNDRECT(VARPTR(L%(0)),15,15,VARPTR
    (P%(0,1)))
450 FOR J=0 TO 3
460 L%(J)=L%(J)-2
470 NEXT J
480 CALL FILLROUNDRECT(VARPTR(L%(0)),15,15,VARPTR
    (P%(0,I+2)))
490 CALL FRAMEROUNDRECT(VARPTR(L%(0)),15,15)
500 CALL MOVETO(E+40,D+13):PRINT TS(I)
510 NEXT I
520 IF MOUSE(0)=1 THEN RESTORE: GOTO 30 ELSE 520
1000 ' data normalization
1010 T8=X(1):B8=X(1)
1020 FOR I8=2 TO N
1030 IF T8<X(I8) THEN T8=X(I8)
1040 IF B8>X(I8) THEN B8=X(I8)
1050 NEXT I8
1060 M8=(T8+B8)/2
1070 FOR I8=1 TO N
1080 X(I8)=X(I8)/M8
1090 NEXT I8
1100 RETURN

```

Our application uses shadowed bars that have different patterns, and it has a legend of patterns identifying the breweries. It seems to be missing a vertical scale to the left of the bars at first glance, but if you reflect a bit on the display, you will see that this feature is unnecessary if you are interested in *relative performance* of the breweries.

10-140	Same as icon chart above, except we reestablish the 12 patterns that were defined in the pie chart.
150	Perform data normalization, only this time use mid-range formula.
160-270	Pattern definitions.
280	$W = 20$ (width), $H = 100$ (height) of high and low.
290	S is used to help center legend text from top to bottom.
300	X is left edge of first bar.
310	E is position of legend from right edge of bar.
320-510	For $I = 1$ to N , number of bars, DO:
320-380	V is this bar's proportional height, $R\%$ defines coordinates of bar's rectangle for shading with pattern. Bar is filled at line 370, and bar constraints adjusted.
390-400	Draw and fill bar.
400	Redefine X for next bar.
420-470	Establish legend box, shadow it, adjust constraints.
480-490	Fill and frame legend box.
500	Move to appropriate place, print legend.
510	Enddo.
520	Sense mouse. If clicked, restore data, return for another chart.
1000	Data normalization routine this time uses yet another form of adjusting data values. We find the midrange by finding the lowest and highest values, and averaging those two values. Then each X is divided by that mean.

We were sorely tempted to add more applications to this book. We enjoy the simple process of taking an existing program developed on another machine and making it do its tricks on the Macintosh. Also, we have found that the Macintosh's outstanding built-in software makes such conversions a joy. The resulting programs are commonly shorter, run faster, and because of better resolution, provide more accurate graphics. It is enough to introduce these ideas, to discuss them with only sufficient detail that you feel tempted to play with them.

That has been our goal from the outset. If we have stimulated you to take some of them and to stretch them here and shrink them there, so that the result is something you wanted but couldn't quite do, then we have achieved our purpose in writing this book. Computer graphics on a personal computer is, more than anything else, a fun exercise. When it becomes tedium, when it no longer teases you to explore the "what if" of a program's variables, graphics will not have its charm and excitement.

But you're in luck. Graphics will assume its appropriate position in the range of computer activities, and it will forever stay as the most exciting procedure for displaying information on a computer's screen, and on its printer's paper.

We await the day when the Macintosh has more memory, a color display, a compiler BASIC, light pens, and who knows what other goodies to help us interact with our pictorial program output. The Macintosh hardware and software, as it exists, is good. Like everything else it will improve, and we will chuckle at some of the simplistic things we did in this book. Still, we are confident that many of the exercises we performed here will make it easier for us in the future, and we won't regret having spent the time to do them. We hope you feel the same way.

Index

- ALPHABET DRAW program 172
ANGLEDRAW program 166
ANGLEWALK program 140
ANSI 63
APPROACHING STAR program 110
ASTROID program 92
Active window printing 25
Ada 64
Alphabet generator 171
Alphabet 16, 23
American National Standards Institute (ANSI) 63
Annotated artwork 17
Apple menu 13
Apples with ALPHABET DRAW 178
Architectural system 49
Artist's Venn 25
Astroid 91
Athens font 25, 27
Atkinson, Bill 1
- BAR CHART program 208
BASIC 63
BENT SIERPINSKI program 157
BIRTHDAY program 95
Bar charts 207
Beers 198
Bent Sierpinski 156
Bicycle icon 32
Binomial distribution application 79
Birthdays application 95
Bleeding images 32
Breweries 198
Brick wall 7
Brush selection menu 2
Brush shape display, edit 3
Brush 1
Bucket 7
Buenos Aires font 29
Business card 15
- CAD/CAM 197
CALLFILLARC 203
CALL FILLOVAL 202, 206
CALL FILLRECT 209
CALL FILLROUNDRECT 203
CALL FRAMEARC 206
CALL FRAMEOVAL 202, 206
CALL FRAMEROUNDREC 206
CALL HIDECURSOR 192
CALL INITCURSOR 191
CALL MOVETO 96
CALL OBSCURECURSOR 192
CALL PENSIZE 97
CALL SETCURSOR 192
CALL SHOWCURSOR 192
CALL TEXTFONT 205
CALL TEXTMODE 96
CALL statement 63
CALLs to text management routines 117
CARDIOID program 89
CHAIN 61, 66
CIRCLE 122
CIRCLES program 148
CLEAR 68
COBOL 64
COMMON 61, 67
CPU 64
Cairo font illustrated 12
Cairo font 17, 100

- Cardioid 88
- Cartesian coordinates 135
- Case structure 56
- Centered Sierpinski's 153
- Chart applications 197
- Charts, bar 207
- Charts, icon 204
- Charts, pie 201
- Charts, various 27
- Chelmsford font 29
- Chicago font 12
- Clocks 117
- Command key 11
- Compilers 64
- Complex tessellation 145
- Constraint 22, 26
- Coordinates, screen 77
- Copy 13
- Copying a picture 11
- Cursor design 190
- Curtate cycloid 86
- Curves, mathematical 84
- Cut 13
- Cycloid, curtate 86
- Cycloid, prolate 84

- DEF-type statements 68
- DIAMONDWALK program 143
- DIGITAL CLOCK program 123
- DOUNTIL structure 55
- DOWHILE structure 54
- DRAW alphabet 171
- DRAW command modes 162
- DRAW command options 162
- DRAW command syntax 159
- DRAW commands, letter A 175
- DRAW motion commands 160
- DRAW subroutine applications 165
- DRAW subroutine pseudocode 163
- DRAW 135
- Decision structure 53
- Design, program 47
- Diamond pattern with DRAW 170
- Diamond tessellation 142
- Directions for DRAW 160
- Drunkard's walk 139
- Duplicating an image 18

- Egg timer program 132
- ENLARGE STAR program 104
- EVOLUTE OF ELLIPSE program 91
- Edit brush shape display 3
- Edit brush shape 1
- Edit brush shapes 8
- Egyptian hieroglyphics 12
- Eraser 9, 16, 22
- Evolute of ellipse 90

- FILLARC 203
- FILLOVAL 202, 206
- FILLRECT 209
- FILLROUNDRECT 203
- FORTTRAN 64
- FRAMEARC 206
- FRAMEOVAL 202, 206
- FRAMEROUNDREC 206
- Fatbits 18, 32
- File menu 9
- Filling patterns 200
- Fish scales 7
- Flip horizontal 19
- Flip vertical 19
- Flipping an image 17
- Font application 29
- Font production 31
- Fonts illustrated 30
- Fontsize menu 12
- Four-pointed star tessellation 143
- Fractals 152
- Frog icon design 100
- Full screen filing 25
- Full screen printing 25

- GET 97
- GWBasic 135
- Gantt charts 27
- Geneva font 12
- Goodies menu 16
- Goodies 21
- Grabber 18, 23, 25
- Graffiti wall 7
- Grid 16, 23, 33

- HIDECursor 192
- HIPO chart 26, 195

- Hierarchy chart 49
- Hierarchy of Input-Process-Output (HIPO)
 - chart 26
- Hieroglyphics 12
- Hipo chart 26
- Hot spot 191
- House plan menu system 57
- Hypocycloid of four cusps 91

- ICON CHART program 205
- INITCURSOR 191
- INVOLUTE OF CIRCLE program 87
- Icon application 31
- Icon charts 204
- Icon ideas list 34
- Icons 29
- Ideograms 204
- Image resolution 77
- Input design 48
- Interpreted code 64
- Involute of circle 86

- LINE STEP 137
- LINE, advanced applications 138
- LINE 135
- Lasso with option key 41
- Lasso 17, 22, 32
- Letter A DRAW commands 175
- Local variables 63
- London font 19
- Loop structures 54
- Los Angeles font 14

- MANTEL CLOCK program 128
- MC68000 64
- MENU program listing 70
- MERGE 61, 67
- MORE CIRCLES program 148
- MOVE instruction, DRAW command 161
- Maintenance, program 51
- Marquee 11, 18
- Mathematical curves 84
- Menu, brush selection 2
- Menus 57
- Modes, DRAW command 162
- Modula-II 64
- Modular programming 45
- Modules, program 49
- Monaco font 12
- Motion commands, DRAW 160
- Mouse tans 181
- Mouse, user interaction with 70
- Moving a picture 11

- NEWS ROOM CLOCK program 129
- New York font 12
- No bicycling icon 32

- OBSCURECURSOR 192
- Olympic icons 31
- Option key 11, 15
- Options, DRAW command 162
- Organization charts 27
- Output design 47
- Overlapping circles 21
- Overlay 67

- PACHINKO program 82
- PEACHES program 74
- PEARS program 74
- PERT chart 195
- PERT charts 27
- PIECHART program 202
- PLUMS program 75
- POINT 79, 135
- PROLATE CYCLOID program 85
- PSET 78, 135
- PTAB 79
- PUT 97
- Pachinko game 79
- Paint bucket 18
- Palette 1
- Pascal 64
- Paste 13
- Pattern definition 200
- Pencil to erase 18
- Pencil 1
- Pfruits 72
- Pie charts 201
- Pixel 18
- Pixels and resolution 77
- Plums 73
- Posters and flyers 33
- Print Draft 25

- Probability of same birthdays 95
- Processes design, programming 48
- Processes, DRAW subroutine 164, 165
- Program ALPHABET DRAW 172
- Program ANGLEDRAW 166
- Program ANGLEWALK 140
- Program APPROACHING STAR 110
- Program ASTROID 92
- Program BAR CHART 208
- Program BENT SIERPINSKI 157
- Program BIRTHDAY 95
- Program CARDIOID 89
- Program CIRCLES 148
- Program DIAMONDWALK 143
- Program DIGITAL CLOCK 123
- Program EGG TIMER 132
- Program ENLARGE STAR 104
- Program EVOLUTE OF ELLIPSE 91
- Program ICON CHART 205
- Program INVOLUTE OF CIRCLE 87
- Program MANTEL CLOCK 128
- Program MENU 70
- Program MORE CIRCLES 148
- Program NEWS ROOM CLOCK 129
- Program PACHINKO 82
- Program PEACHES 74
- Program PEARS 74
- Program PIECHART 202
- Program PLUMS 75
- Program PROLATE CYCLOID 85
- Program RACING STARS 106
- Program RAW DATA 198
- Program REVOLVING STARS 112
- Program ROSES 94
- Program SHOOTING STAR 99
- Program SIERPINSKI 150
- Program SQUAREDRAW 168
- Program SQUAREWALK 142
- Program STARS & CIRCLES 146
- Program STARWALK 144
- Program TAN APPLICATION 185
- Program TWO CLOCKS 126
- Program VINYL FLOORWALK 145
- Program WALL CLOCK 120
- Program coding and testing 51
- Program design 45
- Program flowcharts 27
- Program maintenance 51
- Program modules 46, 49
- Program planning 45
- Programming, structured 50
- Programs 57
- Prolate cycloid 84
- Pseudocode for ALPHABET DRAW 174
- Pseudocode, DRAW subroutine 163
- Pseudocode 49
- Pythagorean theorem 161
- Quickdraw CALLs 117
- Quickdraw routines 65, 197
- RACING STARS program 106
- RAW DATA program 198
- REVOLVING STARS program 112
- ROSES program 94
- Racing invitation application 33
- Random walk problem 138
- Random walk with DRAW 165
- Recursion 149
- References on fractals 152
- References on tangrams 43
- Resolution 77
- Roses 93
- Rounded rectangle 26
- Row of Macs 13
- SETCURSOR 192
- SHOOTING STAR program 99
- SHOWCURSOR 192
- SIERPINSKI program 150
- SQUAREDRAW program 168
- SQUAREWALK program 142
- STARS & CIRCLES program 146
- STARWALK program 144
- Scrapbook 13
- Screen coordinates 77
- Screen displays, TAN APPLICATION program 182
- Sequential structure 52
- Show page 24
- Shrink 15
- Shrinking a picture 11
- Sierpinski patterns 149
- Sierpinski's, centered 153
- Spray can 5
- Square tessellation 141
- Stars and circles tessellation 146

- Stars and motion application 97
- Steps in program planning 47
- Storage design 48
- Storage, MacPaint documents 29
- Stork tangram production 41
- Stork with TAN APPLICATION program 184
- Storks, legal and illegal 39
- Structure, case 56
- Structure, decision 53
- Structure, general 60
- Structure, sequential 52
- Structured programming 50
- Structures, loop 54
- Style menu 12, 17
- Submenus 57
- Subprograms 57
- Subroutines 60
- System flowcharts 27
- System modules 65

- TAN APPLICATION program 185
- TAN APPLICATION variables 190
- TEXTFACE 118
- TEXTFONT 118, 205
- TEXTMODE 119
- TEXTSIZE 119
- TWO CLOCKS program 126
- Tan tile production 40
- Tangram animals 42
- Tangram display screen 181
- Tangram history 38
- Tangram production rules 38
- Tangram reference list 43

- Tangrams 37
- Tans BASIC program 181
- Tans 37
- Tessellation with DRAW 168
- Tessellation 138
- Text management routines 117
- Textfaces 117
- Textsizes 117
- Tiles, broken 37
- Toolbox 64
- Top-down design 45, 46, 65, 66
- Topological rigor 37
- Tree & grass 6
- Tree & leaves 5
- Tree & limbs 4
- Tree trunk 3
- Two heads 17

- UPC symbol 16
- Undo 26
- User friendliness 47

- VINYL FLOORWALK program 145
- Variables in TAN APPLICATION 190
- Variables, local 63
- Venice font 12, 33
- Venn diagram 21

- WALL CLOCK program 120
- Waffle tiling 7
- Waltham font 29
- Webbing 7



EYE-CATCHING ART FOR PROFESSIONAL-LOOKING COMPUTER GRAPHICS!

With ideas to trigger striking displays, this graphics art guide helps artists and managers new to Macintosh graphics create eye-catching art on the Macintosh.

Packed with graphics applications for business data presentations, the book shows you how to use MacPaintTM, so you can create dazzling designs and . . .

- memorable logos
 - unique business cards
 - distinctive typefaces
 - personal or business letterheads
-